CS-1993-16

Implementation Considerations for Higher-Order Features in Logic Programming

Gopalan Nadathur
Bharat Jayaraman
Debra Sue Wilson

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

June 1993

Implementation Considerations for Higher-Order Features in Logic Programming*

Gopalan Nadathur
Department of Computer Science
Duke University, Durham, NC 27706
gopalan@cs.duke.edu

Bharat Jayaraman
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
bharat@cs.buffalo.edu

Debra Sue Wilson
Department of Computer Science
Duke University, Durham, NC 27706
dsw@cs.duke.edu

Abstract

This paper examines implementation problems that arise from providing for aspects of higher-order programming within and enhancing the metalanguage abilities of logic programming. One issue of concern is a representation for the simply-typed lambda terms that replace the usual first-order terms as data structures; this representation must support an efficient realization of λ-conversion operations on these terms. Another issue is the handling of higher-order unification that becomes an integral part of the computational model. An implementation must cater to the branching nature of this operation and also provide a means for temporarily suspending the solution of a unification problem. A final issue concerns the treatment of goals whose structure is not statically apparent. These problems are discussed in detail and solutions to them are described. A representation for lambda terms is presented that uses the de Bruijn "nameless" notation and also permits reduction substitutions to be performed lazily. This notation obviates α-conversion and also supports an efficient implementation of β-reduction. Branching in unification is implemented by using a depth-first search strategy with backtracking. A structure that is called a branch point record and is akin to the choice point record of the Warren Abstract Machine (WAM) is described for remembering alternatives in unification. An explicit representation for unification problems is presented that permits sharing and also supports the rapid reinstatement of earlier versions of the problem. The implementation of unification is tuned to yield an efficient solution to first-order like problems, in fact through the use of compiled code as in the WAM. A compilation method is also discussed for goals whose structure changes during execution. The various ideas presented here form part of an ongoing implementation of a higher-order logic programming language called λProlog.

* This paper has been submitted for publication. Comments on its contents are welcome and may be sent to the first author at the indicated address.

Running Title: Higher-Order Features in Logic Programming

1 Introduction

This paper examines techniques relevant to the implementation of the logic programming language λProlog [30]. The basis for this language is provided by a polymorphic version of the theory of higher-order hereditary Harrop or *hohh* formulas [28]. At a qualitative level, this theory represents an amalgamation of two different extensions to the theory of Horn clauses that underlies Prolog. The extension in one direction is obtained by including higher-order features — in the form of quantification over function and some occurrences of predicate variables and the replacement of first-order terms by simply typed lambda terms — within Horn clauses, resulting in *higher-order* Horn clauses [31]. Along the other direction, Horn clause logic is enhanced by permitting universal quantifiers and restricted uses of implications, resulting in a first-order version of hereditary Harrop formulas [23, 28]. The combination of these two logics produces a logic that is simply typed. This typing paradigm is somewhat constraining from the perspective of programming, but can be relaxed through the introduction of polymorphism. The resulting logic is what constitutes the basis for λProlog.

The enrichments to Horn clause logic that are embodied in the logic underlying λProlog provide for new features at a programming level. λProlog is in fact a language that manifests these features and consequently has several novel capabilities in comparison with a language like Prolog. The usefulness of these capabilities has lead to a significant interest in the language and several systems have been developed that implement λProlog or a close relative of it [4, 10, 11, 27]. These systems notwithstanding, a detailed consideration has not been given in the past to the design of implementation techniques that are well-suited to such a language[1]. The discussions in this paper are part of an effort that focuses on precisely this issue, with the ultimate goal of providing an efficient and robust implementation for λProlog. The hierarchy of logics outlined above has been a useful structuring device in this endeavor. In particular, we have developed an implementation scheme for the full language by starting with the architecture of the machine usually employed in Prolog implementations, and considering independently the new devices that are required for dealing with higher-order aspects, types and implication and universal quantifiers. We believe that adopting such an approach is well justified: unification and backtracking are central to the implementation of all the logics in question, and the usual Prolog machine, exemplified by the Warren Abstract Machine (WAM) [39], provides a good framework for an efficient treatment of these aspects[2]. Furthermore, the new features in λProlog are in a sense orthogonal to each other. Consequently there is little interference between the mechanisms developed for each of these features, and in fact they blend together well in an overall machine [20].

In keeping with the above strategy, this paper discusses implementation methods for one of the new features in λProlog, namely, the inclusion of higher-order notions. It complements, in this respect, other work that we have done concerning the treatment of implications and universal quantifiers in goals [19, 29] and types [21]. The higher-order additions that we consider here allow for a quantification over predicate and function variables and for the use of (typed) lambda terms

[1]The discussions in [4] are in a sense an exception: issues such as memory management are considered there and certain optimizations are indicated especially with regard to implementing reduction on lambda terms. However, even these considerations are not detailed enough to the extent of yielding an abstract machine for λProlog.

[2]See also the comments in [5] to this effect.

instead of first-order terms. The language that results from these additions is of interest in its own right. In the programming context, predicate quantification is the basis for writing procedures that take other procedures as arguments. Including this ability thus provides for aspects of higher-order programming. In a similar sense, the use of lambda terms results in data structures that permit syntactically complex objects like formulas, programs, proofs and types to be represented in a rather natural fashion [7, 18, 26, 36, 38]. Allowing these terms into our language and using the appropriate unification operation on these terms leads to novel metalanguage capabilities, and these have been explored in several places (*e.g.*, see [13, 25, 26, 37]). From an implementation perspective, the new problems posed by our language arise mainly from the presence of lambda terms and the various operations that have to be performed on them. One such operation is that of unifying pairs of lambda terms, commonly referred to as higher-order unification. This operation is much more complex than first-order unification. Performing it may involve a branching search and an adequate implementation must permit this kind of search to be conducted. It may also be necessary at times to "suspend" this operation before a unifier is found, and an implementation must therefore be able to encapsulate partially solved unification problems. At a lower level, an operation that is ubiquitous, and hence whose efficient implementation is crucial, is the reduction of lambda terms. This operation is needed for comparing terms in our context and has special requirements as a consequence: for example, it may be necessary to reduce terms embedded within abstractions, an aspect avoided in usual implementations of reduction. The comparison of terms must also ignore the particular names used for bound variables and the chosen representation for lambda terms must allow this requirement to be factored in efficiently.

We consider the problems outlined above at length in this paper and describe mechanisms that can be incorporated into the basic architecture of the WAM towards overcoming them. We propose a representation for lambda terms that provides a suitable basis for implementing a language such as ours. This representation uses the de Bruijn notation [6] for lambda terms which obviates renaming operations in comparing them. We enhance the de Bruijn notation with a means for explicitly encoding substitutions in terms, thereby permitting substitutions arising out of reduction steps to be performed lazily. Our notation leads to efficiency in reduction and also permits a postponing of work that might later be discovered to be unnecessary. Further improvements to this notation are considered that reduce the substitution work and also facilitate a sharing in reduction steps. Implementations of various necessary operations on terms are outlined based on this notation. With regard to reduction, our implementation is conservative in its use of space in addition to time, a factor that earlier experimental implementations of λProlog show to be important. We also develop mechanisms for supporting the special needs of higher-order unification. We present a method for explicitly representing pairs of terms that need to be unified. Our representation fosters sharing among incrementally changing collections of such pairs and uses ideas germane to Prolog implementations for rapidly reinstating of sets of such pairs that were in existence earlier. Branching in higher-order unification is handled within our implementation by conducting a depth-first search, and we introduce the idea of a *branch point* record as a means for encoding unexplored alternatives in the state of our machine. These new records are akin to the choice point records of the WAM and similarly enable a rapid return to an earlier state and followed by the choice of an alternative search path. Although branching in unification may eventually be necessary, experimental evidence suggests that it might often be avoided by some simple processing steps [22]. Our implementation is sensitive to this fact at several levels. First, the processing structure permits the easy application of such steps. Second, the creation of branch point records and the explicit encoding of unification problems is delayed until after these steps have been applied. Third, specific operations are considered towards eliminating branching. With regard to the last aspect, our implementation permits a treatment of first-order like unification problems through the usual mechanisms of the WAM and can deal with these problems almost entirely through compiled code.

The remainder of this paper is structured as follows: In the next section we motivate the higher-order additions that we eventually consider through a sequence of examples. These examples are presented informally in a language with Prolog-like syntax. We make this language precise in Section 3 by describing a higher-order version of Horn clauses and then highlight some of the problems that arise in its implementation. An understanding of higher-order unification is necessary for dealing with these problems. We outline the procedure of Huet [17] in Section 4 and use this in describing an abstract interpreter for our language. We then consider an actual implementation of this interpreter. In Section 5, we discuss issues determining a satisfactory representation of lambda terms and distill from this the representation that we eventually use in our implementation. The special requirements for higher-order unification are dealt with in Section 6: we describe here a representation for (suspended) unification problems, present an encoding of branching in unification and integrate our representation of terms into the unification process. We incorporate the new structures into the overall architecture of the WAM in Section 7, focussing largely on the interactions of these two components. Section 8 is concerned with the possibility of compilation within our implementation scheme. The focus here is on compiling first-order like unification, and we outline changes and additions to the usual WAM instructions necessary for our language. Goals that have predicate variables as their heads are an integral part of our language and a complete implementation must include a treatment of such goals. Fortunately, this turns out to be simple and we indicate a method for compiling such goals in Section 8. This section also contains examples illustrating the compilation ideas as well as the behavior of the overall machine. Section 9 is a conclusion to this paper.

2 Higher-Order Features in Logic Programming

We assume a familiarity with the syntax and semantics of Prolog described, for example, in [8]. Our purpose here is to expose certain kinds of higher-order features that might be added to this language. We attempt to motivate such additions and to describe informally the syntax and semantics of the resulting language. However, our discussions will not be comprehensive in either respect. The usefulness of higher-order features is examined in several other places (*e.g.*, [12, 16, 25, 26, 37]) and a logic underlying the proposed extension will be presented more precisely in the next section.

A higher-order ability that is usually coveted in programming contexts is that of passing procedures as arguments to other procedures and of returning these as the results of computations. Indeed, within the functional programming paradigm this is the defining characteristic of a higher-order language. Towards realizing such an ability in a logic programming language we observe first the duality between predicates and procedures in this context. We further note the uniform manner in which parameters are passed and values are returned: both are achieved by unifying the arguments of predicates. From a logical perspective, realizing the desired higher-order abilities thus hinges on the possibility of quantifying over predicates.

Predicate quantification is, in fact, capable of providing at least the most rudimentary of higher-order abilities. A typical illustration of this may be provided through a procedure for "mapping" a function over a list. The following clauses that define the procedure *mappred* realizes this function:

$mappred([\,], P, [\,])$.
$mappred([X|L1], P, [Y|L2])$:- $P(X, Y), mappred(L1, P, L2)$.

Notice that the variable P that appears in these clauses is predicate variable that is implicitly universally quantified over the clause. From a programming viewpoint, this variable permits procedures to be passed as arguments: it can be instantiated with a procedure name and would lead to the invocation of that procedure (with appropriate arguments) in the course of evaluating a *mappred* query. As a specific example, assume that the program also contains the following list of clauses defining the ages of various individuals:

$age(bob, 24)$.
$age(sue, 23)$.

An invocation of *mappred* may then be performed through the query

?- $mappred([bob, sue], age, L)$.

Solving this query results in the invocation of queries of the form $age(bob, Y1)$ and $age(sue, Y2)$ and eventually leads to the instantiation of L to $[24, 23]$.

The above example involves a simple kind of instantiation for predicate variables — the substitution of a name of a predicate. More sophisticated instantiations can arise if argument terms are permitted to have a richer structure. For example, there is often a need to have a "view" of a procedure that is based on changing the order of its arguments. Such a view would be necessary for the *age* predicate if we were to use it together with *mappred* to generate a list of individuals from a list of ages. Such views can be easily created if the terms of a lambda calculus are permitted as arguments. Thus, a relation that is like *age* but has its arguments reversed is given by the predicate term $\lambda x\, \lambda y\, age(y, x)$. This term can be used to form a query that computes a list of individuals from a list of ages:

?- $mappred([24, 23], \lambda x\, \lambda y\, age(y, x), L)$.

In order to solve this query the computational machinery for our language must be able to perform reductions on lambda terms. Thus, a term such as $(\lambda x\, \lambda y\, age(y, x))(24, Y1)$ will have to be transformed into $age(Y1, 24)$ before it is invoked as a query. Another kind of relation that lambda terms might be used to generate is that obtained by fixing the values of certain arguments of given relations. For example, a representation of the set of individuals whose age is 24 can be obtained from the *age* predicate by using the term $\lambda x\, age(x, 24)$.

The usefulness of lambda terms can be further enhanced by allowing logical connectives and quantifiers to appear in these terms. To see some of the benefits of this, let us assume that our program contains a list of clauses defining the "parent" relation between individuals:

$parent(bob, john)$.
$parent(john, mary)$.
$parent(sue, dick)$.
$parent(dick, kate)$.

Given these clauses, a grandparent relation can be created by the term

$\lambda x\, \lambda y\, \exists z\, (parent(x, z) \wedge parent(z, y))$.

This term uses the existential quantifier and a conjunction to effectively "join" two relations in obtaining a new one. It may, for instance, be used in conjunction with the *mappred* relation in querying for the grandparents of the individuals in a given list:

$$\texttt{?-}\ mappred([bob, sue], \lambda x\, \lambda y\, \exists z\, (parent(x, z) \wedge parent(z, y)), L).$$

Solving this query requires solving a new one with an existential quantifier and a conjunction. Clearly an implementation of our language must be capable of evaluating queries of this sort that arise dynamically.

A question that arises is that of what logical symbols should be permitted to appear in terms. The rationale for allowing logic into terms in the first place provides a basis for answering this question. Predicate terms are designed to instantiate predicate variables that get invoked as queries after being supplied with appropriate arguments. The logical connectives and quantifiers that appear in terms therefore have the potential of becoming the top-level logical symbols of goals. Thus, only those logical symbols that can appear at this level in Prolog goals should be allowed to appear in terms. In particular, only conjunctions, disjunctions and existential quantifications should be permitted.

Like predicate quantification, quantification over function variables might also be considered. While predicate variables permit customary higher-order features to be realized within logic programming, function variables lead to some truly novel computational abilities. Towards understanding this aspect, let us first consider the following "functional" counterpart of *mappred*:

$$mapfun([], F, []).$$
$$mapfun([X|L1], F, [F(X)|L2])\ \texttt{:-}\ mapfun(L1, F, L2).$$

The predicate *mapfun* defined by these clauses relates a function and two lists just in case the second list is obtained by applying the function to each element of the first list. However, the notion of function application is based on reduction in the lambda calculus and is quite different from the idea of evaluating a query. As an example, the query

$$\texttt{?-}\ mapfun([a, b], \lambda x\, g(a, x), L)$$

would be evaluated by applying the term $\lambda x\, g(a, x)$ to each of a and b, eventually producing the list $[g(a, a), g(a, b)]$ as an instantiation for L.

It is also possible to run queries such as the one above in "reverse." For example, the query

$$\texttt{?-}\ mapfun([a, b], F, [g(a, a), g(a, b)])$$

might be posed with the expectation of generating the substitution $\lambda x\, g(a, x)$ for F. In reality computations such as this can feasibly be carried out only in a context where all terms are typed and would require unification of terms to be based on a notion of equality the assumes the rules of lambda conversion. Furthermore, the nature of the computation would be somewhat different from the one usually present in Prolog, possibly involving backtracking over the choice of substitutions. Thus, in the evaluation of the above query, the first significant unification problem that is considered is that of making $F(a)$ and $g(a, a)$ identical. There are four "most general" unifiers given by the substitutions $\lambda x\, g(x, x)$, $\lambda x\, g(a, a)$, $\lambda x\, g(x, a)$ and $\lambda x\, g(a, x)$ for F. At a subsequent stage, the terms $F(b)$ and $g(a, b)$ will have to be unified. If a substitution for F other than the last one had been chosen initially, this pair of terms cannot be unified. Backtracking would have to be initiated in this case leading to an alternative substitution being picked. An implementation of the language must, of course, incorporate the necessary mechanisms for recording choices even in substitutions.

The two *mapfun* queries are useful in understanding the new programming abilities provided by function variables. In both examples, function variables and lambda terms are used in analyzing the process of substitution. In the first query, the computation involves affecting substitutions into a given structure, namely $g(a, X)$. This structure could have been much more complex, involving additional bound variables. Substitution becomes complicated in this case because there is a potential for free variables in the term being substituted to be inadvertently captured by abstractions. However, a proper implementation of lambda reduction automatically ensures that this is done correctly. The second query involves finding a structure from which two different structures can be obtained by substitutions; in particular, a structure which gives $g(a, a)$ if a is substituted into it and $g(a, b)$ if b is used instead. The operation of higher-order unification provides a means for answering such questions, and the substitution that is found — $\lambda x\, g(a, x)$ — represents an appropriate answer.

More generally, function variables and lambda terms are useful in representing and manipulating objects that naturally incorporate the notion of binding. One category of such objects is that of logical formulas. Thus, let us consider representing the formula $\forall x\, (p(x) \vee q(x))$. The quantifier in this formula fulfills two functions: it determines a scope for the variable x and it makes a predication of the resulting structure. These two functions can be separated and the former can be represented by lambda abstraction. Using this idea, our formula can be represented by the lambda term $all(\lambda x\, or(p(x), q(x)))$, where *all* and *or* are constants that are used to encode the logical symbols. Such a representation enables a simple implementation of several operations that are typically found within a formula manipulating system. For example, it usually necessary in such a system to recognize that the names of bound variables are unimportant. This fact is reflected in our representation by the equality of expressions such as $all(\lambda x\, or(p(x), q(x)))$ and $all(\lambda y\, or(p(y), q(y)))$. Similarly, it might be necessary to instantiate universally quantified formulas. Given our representation, this is readily implemented by lambda conversion. Thus, instantiating $\forall x\, (p(x) \vee q(x))$ with the term $f(a)$ is simply expressed by the application $(\lambda x\, or(p(x), q(x)))(f(a))$; this term reduces to the representation of the desired instance, $or(p(f(a)), q(f(a)))$. The clause

$$instan(all(P), T, P(T)).$$

in fact implements this instantiation relation in our language.

Unification is also a useful operation in the context of our representation. Using the clause above provides one example: the "formula" to be instantiated must first be unified with $all(P)$. This unification problem has the flavor of a first-order one even though a higher-order variable, P, might be instantiated. However, genuine higher-order unification can also be used, resulting in sophisticated pattern matching capabilities. As one example, the term $all(\lambda x\, or(P(x), Q(y)))$ can be used as a template for recognizing formulas containing a top-level universal quantifier whose scope is a disjunction; P and Q are schema (or logic) variables in this template. If our desire is to recognize a formula of this kind in which the second disjunct does *not* depend on the quantifier, we may use the template $all(\lambda x\, or(P(x), Q)$ instead. Recognizing such properties may actually be useful: for instance, this particular one is the basis for moving a universal quantifier over a disjunction in the course of constructing a proof.

Abstracting out of the preceding discussion, we see that function variables, lambda terms and higher-order unification are useful in writing programs that manipulate objects whose structure naturally involves a notion of binding. There are several objects of this kind in addition to formulas: programs, proofs, types and even constructs in a natural language. Thus a language such as ours can be useful in programming tasks in realms concerning these objects. This potential has been confirmed by several experiments conducted with such a language; see [13, 15, 25, 26, 37] for some examples.

Based on the treatment of function variables, one may wonder if values can also be found for predicate variables. At one level, we may ask what answer should be given to the query

$$mappred([bob, sue], P, [24, 23]).$$

Given the clauses defining *age* and *mappred* from earlier in this section, it may appear that the appropriate answer to this query is the value *age* for P. However, a little reflection reveals that this query is really an ill-posed one. There are too many properties that are true of *Bob* and 24 and *Sue* and 23; indeed any property of the form $\lambda x\, \lambda y\, Q$, where Q is a true predicate suffices. There is actually one substitution for P that is a solution that in a sense includes all others. This is the universal relation given by $\lambda x\, \lambda y\, \top$, where $\top$ represents the tautologous proposition. The implementation we consider later will produce only this solution.

Notwithstanding the above example, meaningful values can be computed for predicate variables. In particular, unification can be used to find such values when these variables are used in the capacity of function variables. Predicate variables thus have a dual role: they can be used in terms and as arguments of atomic formulas and values can be computed for them when they are used in this fashion, and they can be used as the heads of atomic predicates and in this case they give rise to queries that can be evaluated. This duality can be exploited to perform interesting computations. For example, consider the program consisting of the following clauses:

$$primrel(mother).$$
$$primrel(wife).$$
$$rel(R) \;\texttt{:-}\; primrel(R).$$
$$rel(\lambda x\, \lambda y\, \exists z\, (R(x, z) \wedge S(z, y))) \;\texttt{:-}\; primrel(R), primrel(S).$$
$$mother(jane, mary).$$
$$wife(john, jane).$$

Given this program, the query

$$\texttt{?-}\; rel(R), R(john, mary)$$

succeeds only if the substitution $\lambda x\, \lambda y\, \exists z\, (wife(x, z) \wedge mother(z, y))$ is made for R. The behavior exhibited here is similar to that of function producing functions in other programming paradigms: *rel* is used to produce a procedure through simple computations and this procedure is then applied to some arguments. Notice that reversing the goals, *i.e.*, using the query

$$\texttt{?-}\; R(john, mary), rel(R)$$

instead should also produce the same answer. However, ensuring this requires delaying the first goal till R has been sufficiently instantiated. The implementation we discuss will not do this, and will choose to solve the first goal eagerly. Thus the substitution $\lambda x\, \lambda y\, \top$ will be produced for R and the reordered goal will actually fail.

3 Higher-Order Horn Clauses

We now describe a logical language that incorporates the various higher order features outlined in the previous section. For this purpose, we need the terms of a lambda calculus, and we use the simply typed lambda terms of [7]. An integral part of these terms is a notion of types. The relevant types are constructed from a set $\mathcal{S}$ of sorts by using a set $\mathcal{C}$ of type constructors, each specified with a unique arity, and the function type constructor $\rightarrow$. We assume here that $\mathcal{S}$ contains the sorts i and o for individuals and booleans and that $\mathcal{C}$ contains the unary list type constructor *list*. These sets can be added to by the user of the language in some manner whose details are not currently relevant. The collection of types is now the smallest set satisfying the following properties: (i) each member of $\mathcal{S}$ is a type, (ii) if $\alpha_1, \ldots, \alpha_n$ are types and $c \in \mathcal{C}$ is of arity n, then $(c\ \alpha_1\ \ldots\ \alpha_n)$ is a type, and (iii) if α and β are types, then so is $(\alpha \rightarrow \beta)$. A type is an *atomic* type if it is obtained by using (i) or (ii) above and is a *function* type otherwise. We assume that $\rightarrow$ is right associative. A function type may thus be written in the form $\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta$ where β is an atomic type; β is then said to be its *target* type and $\alpha_1, \ldots, \alpha_n$ are its *argument* types. This notation and terminology is extended to atomic types by permitting the argument types to be an empty sequence. As illustrations of these various notions, o and $(list\ i)$ are atomic types and $(list\ i) \rightarrow o$ is a function type. The last type has $(list\ i)$ as its argument type and o as its target type.

The terms and their types are obtained from denumerable sets of typed variables and constants using the following rules: (i) each constant and variable of type α is a term of type α, (ii) if x is a variable of type α and t is a term of type β, then $(\lambda x\, t)$ is a term of type $(\alpha \rightarrow \beta)$ and is called an *abstraction* that *binds* x and has *scope* t, and (iii) if t_1 and t_2 are terms of type $(\alpha \rightarrow \beta)$ and α respectively, then $(t_1\ t_2)$ is a term of type β and is called an *application* of t_1 to t_2. The constants are partitioned into *parameters* and *logical* constants. The latter set consists of the symbols $\top$ of type o, $\sim$ of type $o \rightarrow o$, $\wedge$, $\vee$ and $\supset$ of type $o \rightarrow o \rightarrow o$ and, for each α, $\exists_\alpha$ and $\forall_\alpha$ of type $(\alpha \rightarrow o) \rightarrow o$. These symbols stand for the tautologous proposition, negation, conjunction, disjunction, implication and the (generalized) existential and universal quantifiers, respectively. In writing terms, we will usually drop the type subscripts on $\exists$ and $\forall$ and also abbreviate expressions of the form $(\exists\ (\lambda x\, B))$ and $(\forall\ (\lambda x\, B))$ by $\exists x\, B$ and $\forall x\, B$ respectively. Further, we will write $\wedge$, $\vee$ and $\supset$ in the usual infix manner and we will assume that abstraction and application are, respectively, right and left associative.

The notions of free and bound variables and of subterms of a term should be apparent from the formation rules described above. An important notion is that of equality between terms. In the usual first-order context, two terms are considered equal just in case they are identical. For our lambda terms, this relation must also incorporate the rules of lambda conversion. Towards explaining these rules, let us say that a term s is free for the variable x in the term t if x does not appear free in t in the scope of an abstraction that binds a free variable of s. Further, let $[s/x]t$ represent the result of replacing all the free occurrences of x in t by s. The relevant rules are then the following:

(i) (α-conversion) Replacing a subterm of the form $(\lambda x\, t)$ in a given term by $(\lambda y\, [y/x]t)$, provided y is free for x in t.

(ii) (β-conversion) Replacing a subterm of the form $((\lambda x\, t_1)\ t_2)$ in a given term by $[t_2/x]t_1$ or vice versa, provided t_2 is free for x in t_1.

(iii) (η-conversion) Replacing a subterm of the form t in a given term by $\lambda x\,(t\ x)$ and vice versa, provided t is of type $\alpha \rightarrow \beta$, x is of type α and x does not appear free in t.

We consider two terms to be equal in our context if one can be obtained from the other by using a sequence of these rules. We will need to consider the idea of unifying two lambda terms. The interest here is in substituting terms of matching types for free variables so that the two terms become equal in the sense just outlined. This operation must be performed with care to ensure that free variables in the substituted terms do not get accidentally bound in the result. The correct characterization of this substitution operation can be provided by the lambda conversion rules themselves: assuming that, for $1 \leq i \leq n$, t_i and x_i are a term and a variable of identical type, the result of substituting these terms for the corresponding variables in t is given by the term $((\lambda x_1 \ldots \lambda x_n\, t)\ t_1\ \ldots\ t_n)$.

The attempt to unify terms will actually involve *determining* equality between terms. Fortunately there exist normal forms for terms that can be used for this purpose. To obtain these normal forms, we need to consider a directed application of the rules above. This idea is made precise as follows. First, we identify a subterm of a term of the form $((\lambda x\, t_1)\ t_2)$ as a *β-redex*. The condition permitting the replacement of this subterm as per rule (ii) may not be satisfied in general, but this can be corrected by using a sequence of α-conversion steps. We call such a sequence followed by the desired application of rule (ii) a β-contraction. In a similar sense, a subterm of the form $(\lambda x\ (t\ x))$ in which x does not appear free in t is referred to as an η-redex and a replacement of this subterm by t is called an η-contraction. Finally, we call a sequence of α-conversions and β-contractions a β-reduction, and if the sequence also includes η-contractions it is referred to as a λ-reduction.

In the informal discussion of higher-order features, we had assumed that the only logical symbols permitted in the arguments of atomic formulas were $\wedge$, $\vee$ and $\exists$. This restriction is obtained formally by identifying a class of terms called the *positive* terms in which only these logical constants are allowed to appear. In identifying a class of boolean terms that are the formal counterparts of queries in the language of the previous section, we need a suitable notion of atomic formulas or atoms. We choose these to be terms of type o of the form $(P\ t_1\ \ldots\ t_n)$ where P is a (predicate) variable or parameter and the t_is are positive terms. An atom is said to be *rigid* just in case P, its predicate head, is a parameter and to be *flexible* otherwise. We denote arbitrary atoms by A and rigid atoms by A_r below. *Goal* formulas, denoted by G, are then boolean terms whose structure is given by the syntax rule

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x\, G.$$

These formulas are the desired analogues of queries or goals in Prolog and incorporate the higher-order features discussed: the arguments of atomic goals are lambda terms and predicate and function variables are permitted in goals. A higher-order Horn clause or program clause is the universal closure of a term of the form A_r or $G \supset A_r$. These formulas are the analogues of the usual notion of program clauses. The restriction to rigid atoms is well-motivated: a program clause is to be construed as a partial definition of a procedure whose name is the head of this atom and this can be done meaningfully only if the head is something that can be thought of as a name.

A set of (higher-order) program clauses constitutes a program. As an example, let *nil*, *cons* and *mapfun* be parameters of type $(list\ i)$, $i \rightarrow (list\ i) \rightarrow (list\ i)$ and $((list\ i) \rightarrow (i \rightarrow i) \rightarrow (list\ i) \rightarrow o)$ respectively. Then the clauses

$\forall F\,(mapfun\ nil\ F\ nil)$ and
$\forall X\,\forall F\,\forall L1\,\forall L2\,((mapfun\ L1\ F\ L2) \supset (mapfun\ (cons\ X\ L1)\ F\ (cons\ (F\ X)\ L2)))$

constitute a program. Using some of the conventions of Prolog — *i.e.*, its list notation, its suggestive manner of writing implications, and its convention of leaving quantifiers implicit and denoting implicitly quantified variables by names beginning with an uppercase letter — these clauses can be rendered into the following "friendlier" syntax:

$(mapfun\ []\ F\ [\,])$.
$(mapfun\ [X|L1]\ F\ [(F\ X)|L2])$:- $(mapfun\ L1\ F\ L2)$.

Written in this form, the similarity to the "definition" of $mapfun$ in Section 2 is evident. The differences are the use of a curried notation for atoms and the presence of typing. We adopt this kind of a syntax in the discussions below.

From a logical perspective, the idea of answering a query can be explained by using provability in classical logic and this aspect is investigated in [31]. Operationally, this results in a recipe for solving a closed query from a program $\mathcal{P}$ that is based on the structure of the query:

(1) Solve $G_1 \wedge G_2$ by solving both G_1 and G_2.

(2) Solve $G_1 \vee G_2$ by solving one of G_1 and G_2.

(3) Solve $\exists x\, G$ by solving $[t/x]G$ for some closed positive term t.

(4) Solve a rigid atom A by doing one of the following:

   (a) determining that it is equal to a ground instance of a clause in $\mathcal{P}$, or

   (b) by finding a ground instance $G \supset A'$ of a clause in $\mathcal{P}$ such that A and A' are equal and then solving G.

In this description, a ground instance of a program clause is generated by substituting closed positive terms for the universally quantified variables in the clauses. Not included here is the method for solving $\top$ or a flexible atom. The former is solved immediately. The latter kind of goal does not arise within the context of this recipe because only closed queries are considered. We will in reality be interested in queries that have free variables in them. Answering such a query consists of solving its existential closure and finding substitutions for the implicitly quantified variables that yield a solution. This view is compatible with the method for solving existential queries in the recipe.

The recipe above explains the semantics of our language. However it is not complete as a description of *how* the computation involved in answering a query is to be carried out. One major problem in this regard is that an oracle is assumed for picking a suitable instantiation of an existentially quantified goal. A related problem is that the recipe is nondeterministic; for example, there is a choice in the disjunct of a disjunctive goal that is to be solved and in the clause that is to be used to solve an atom. However, the instructions in this recipe are similar to the ones that are provided for the usual first-order language, and examining the methods used for implementing them in that context gives us an idea of what might be done here. The problem with existential quantifiers is typically dealt with by delaying the instantiations of such quantifiers till such time that information is available for making an appropriate choice. This effect is achieved by replacing the quantified variables by placeholders whose values are determined through unification. Thus, a goal such as $\exists x\, G(x)$ is transformed into one of the form $G(X)$ where X is a new "logic" variable that may be instantiated at a later stage. This device is "cashed out" when solving an atomic goal. Given such a goal A, we look for a program clause $\forall y_1 \ldots \forall y_n\,(G' \supset A')$ such that A unifies with the formula that results from A' by replacing the universally quantified variables with new logic variables. If a clause of this kind is found, the next task becomes one of solving the resulting instance of G'. With regard to nondeterminism, the solution is make a choice in a predetermined manner and to reconsider the choice only in case of failure. In particular, disjunctive goals are considered in a left-to-right order and program clauses are used in the order of presentation. It turns out that the treatment of the logical connectives, the sequencing through program clauses and much of the unification involved in this process can actually be compiled and this is in fact what is done within machine models such as the WAM.

The ideas discussed above can be used in implementing our higher-order language as well. However, some additional problems that must be dealt in our context: a new kind of operation on terms, namely, lambda conversion, must be implemented, a more complex unification process must be accommodated and methods for dealing with flexible goals in program clauses must be devised. The first two aspects have been highlighted in the last section. The need to deal with these two issues, and especially the need to implement the conversion operations efficiently, require a representation for lambda terms to be chosen carefully. A detailed understanding of what is involved in implementing higher-order unification must await a finer grained analysis of the unification process, which we undertake in the next section. Nevertheless, two broad questions can already be identified: first, how is the branching over unifiers to be dealt with and, second, can aspects of this richer unification operation be compiled? To understand what the final aspect, *i.e.*, that of flexible goals, involves, we may consider the following rendition of the *mappred* predicate into our logical language:

$(mappred\ []\ P\ [])$.
$(mappred\ [X|L1]\ P\ [Y|L2])$:- $(P\ X\ Y), (mappred\ L1\ P\ L2)$.

The type of *mappred* is assumed to be $(list\ i) \to (i \to i \to o) \to (list\ i) \to o$ and conjunction is denoted by comma in these clauses. There is a flexible atomic goal in the body of the second clause. The issue of concern is that of how this goal should be "compiled." Notice that P would be instantiated in a computation and we desire an implementation that deals with any given instantiation in an efficient fashion.

We address these implementation questions in the following sections. Before embarking on this, however, it is relevant to comment on the typing scheme assumed for our language. This scheme is somewhat rigid, being based on simple types. The predicates *mapfun* and *mappred* defined above are, for instance, restricted to apply to lists of individuals and cannot be used with lists of booleans, lists of lists or lists of function objects. This inflexibility can be alleviated by injecting a form of polymorphism through the use of type variables. Thus *mapfun* may have been defined to be of type $(list\ A) \to (A \to B) \to (list\ B) \to o$, where A and B can be instantiated to arbitrary types. This is in fact the course that is adopted in λProlog. We have eschewed this kind of polymorphism here because it raises additional implementation problems that we presently do not wish to consider. However, these problems are considered for a first-order language in [21] and the solutions provided therein are entirely compatible with the implementation ideas we develop for our simply typed language here. Another point to note is that even if simple types are used, there is no real loss in flexibility in comparison with an untyped first-order language. Types are primarily meant to make distinctions between different kinds of functional objects. Thus the untyped first-order language can be mapped into our language by defining all first-order objects to be of the same sort. We use this idea implicitly below, by considering most first-order objects to be of type i.

4 An Abstract Interpreter

We refine the recipe for solving a goal formula into a form that can serve as the basis for implementation. For this purpose, we must consider the higher-order unification problem explicitly. This problem is in the general case given by a *disagreement set*, *i.e.*, a finite set $\{\langle t_1, s_1\rangle, \ldots, \langle t_n, s_n\rangle\}$ of pairs of terms with the two terms in each pair having the same type. A solution to such a problem is a substitution σ that when applied to the the two terms in any pair makes them equal. These solutions are, of course, what constitute unifiers. The search for these unifiers can be factored into the repeated use of certain atomic operations [17]. We outline these operations below and then integrate them into an abstract interpreter for our language.

Every term in our language can be λ-converted into what we call a (weak) head-normal form. This is a term of the form $\lambda x_1 \ldots \lambda x_n\,(A\ t_1\ \ldots\ t_m)$ where A is a constant or variable. We assume henceforth that we are only dealing with terms in this form. Given such a term, A is called its *head* and the term is said to be *rigid* if A is a constant or an element of $\{x_1, \ldots, x_n\}$, and *flexible* otherwise. Now, the first operation used in the unification procedure is based on the following observation. Two rigid terms that are of the same type and have the forms $\lambda x_1 \ldots \lambda x_n\,(A_1\ s_1\ \ldots\ s_i)$ and $\lambda x_1\ \ldots\ \lambda x_n\,(A_2\ r_1\ \ldots\ r_j)$ respectively, are unifiable only if A_1 and A_2 are identical. Further, if A_1 and A_2 are identical, these terms have the same set of unifiers as the disagreement set

$$\{\langle \lambda x_1 \ldots \lambda x_n\, s_1, \lambda x_1 \ldots \lambda x_n\, r_1\rangle, \ldots, \langle \lambda x_1 \ldots \lambda x_n\, s_i, \lambda x_1 \ldots \lambda x_n\, r_i\rangle\}.$$

Given an arbitrary disagreement set, this observation can be used either to conclude that it has no unifiers or to reduce it to another disagreement set with the same unifiers and in which each pair has at least one flexible term. We assume below that this simplification process is carried out by a function called $SIMPL$ that returns either a disagreement set or, in the case that impossibility of unification is detected, the marker **F**. We observe that this operation is similar to the term simplification performed in first-order unification. In the latter case, the abstractions at the front of the term are absent and the head is a (constant) function symbol.

If $SIMPL$ does not detect failure, it produces a disagreement set that either has only "flexible-flexible" pairs or has at least one "flexible-rigid" pair. For the former kind of set, (a) there is at least one unifier and (b) a complete search for unifiers is fairly unconstrained [17]. The useful strategy for such sets is therefore to delay looking for specific unifiers. In the case that the set has at least one flexible-rigid pair, the search can proceed by attempting substitutions for reducing the difference between the terms in this pair. Two kinds of elementary substitutions may be employed for this purpose. The first makes the head of the flexible term "imitate" that of the rigid term, and the second "projects" it onto one of the arguments in the hope that the head of the resulting term may be made identical to the rigid one. In particular, let T_1 be the flexible term with F as its head and let T_2 be the rigid term with c as its head. Further, let the types of F and c be $\alpha_1 \to \cdots \to \alpha_k \to \beta$ and $\gamma_1 \to \cdots \to \gamma_j \to \beta$ respectively. Then

(i) $IMIT(T_1, T_2)$, the imitation substitution, is defined only when c is a constant and is the following:

$$\{\langle F, \lambda w_1 \ldots \lambda w_k\,(c\ (H_1\ w_1\ \ldots\ w_k)\ \ldots\ (H_j\ w_1\ \ldots\ w_k))\rangle\}$$

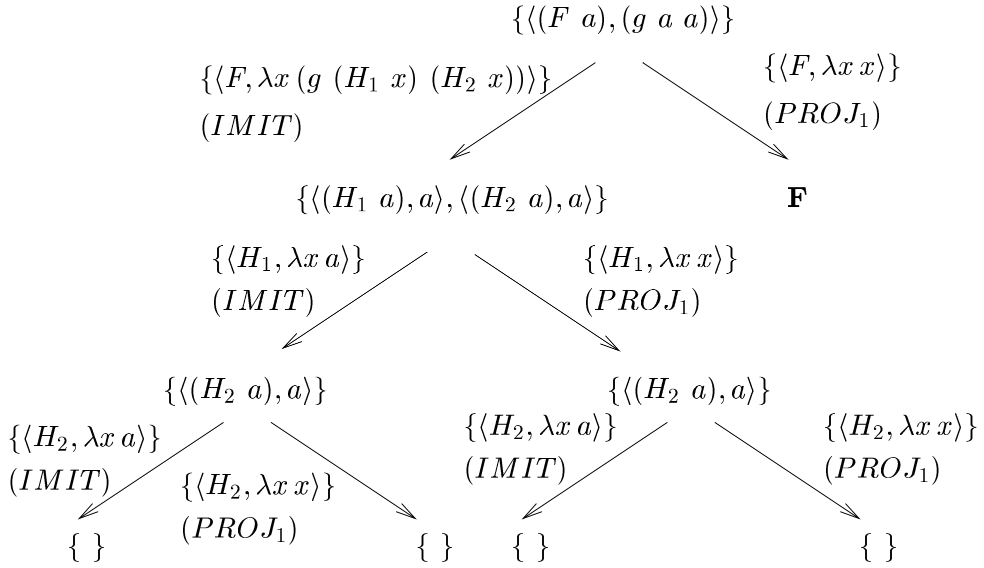


Figure 1: A Matching Tree for $\{\langle (F\ a), (g\ a\ a)\rangle\}$

(ii) for $1 \leq i \leq k$, $PROJ_i(T_1, T_2)$, the i^{th} projection substitution, is defined only when α_i is of the form $\beta_1 \rightarrow \cdots \rightarrow \beta_l \rightarrow \beta$ and is the following:

$$\{\langle F, \lambda w_1 \ldots \lambda w_k\,(w_i\ (H_1\ w_1\ \ldots\ w_k)\ \ldots\ (H_l\ w_1\ \ldots\ w_k))\rangle\}\}.$$

The Hs above are assumed to be new free variables of the appropriate types. We observe that these substitutions are determined entirely by the heads of the flexible and rigid terms.

The two operations described above can be used iteratively in searching for unifiers. The structure of such a search can be depicted by a matching tree [17], illustrated in Figure 1 for the unification problem $\{\langle (F\ a), (g\ a\ a)\rangle\}$ in which a is assumed to be of type i, F of type $(i \rightarrow i)$ and g of type $(i \rightarrow i \rightarrow i)$. The arcs in this tree are labelled with substitutions suggested by $IMIT$ and $PROJ_i$ and the nodes represent the result of transforming the set on the prior node by applying the substitution on the incoming arc and then performing $SIMPL$. The leaves of such a tree are either labelled with **F** or with a (possibly empty) set containing only flexible-flexible pairs. A solution to the original unification problem can be obtained by composing the substitutions on the path to the latter kind of leaf with a unifier for that leaf. In the example presented, these unifiers involve substituting $\lambda x\,(g\ a\ a)$, $\lambda x\,(g\ a\ x)$, $\lambda x\,(g\ x\ a)$ and $\lambda x\,(g\ x\ x)$ for F. A matching tree is exhaustive in the sense that the unifiers of the leaves of a completely expanded tree can be used in this fashion to produce all the unifiers of the original set. In general, such a tree may be infinite and may also include nonterminating branches. Notice however that it must always be finitely branching.

The matching tree is similar in structure to SLD-trees used in conjunction with (pure) Prolog [3]. It is in fact possible to merge these two trees together to get one tree that describes the search space for our higher-order language. This can be done through the notion of a $\mathcal{P}$-derivation [31].

Let $MATCH$ be a function on flexible-rigid disagreement pairs that produces the set of imitation and projection substitutions for any given pair. Let $\mathcal{P}$ be a set of higher-order Horn clauses and let $\mathcal{G}$, $\mathcal{D}$ and θ be symbols for sets of goal formulas, disagreement sets and substitutions respectively. Then the tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2 \rangle$ is said to be *$\mathcal{P}$-derivable* from the tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1 \rangle$ if $\mathcal{D}_1 \neq \mathbf{F}$ and, in addition, one of the following situations holds:

(1) (*Goal reduction step*) $\theta_2 = \emptyset$, $\mathcal{D}_2 = \mathcal{D}_1$, and for some $G \in \mathcal{G}_1$ it is the case that

  (a) G is $\top$ and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$, or
  (b) G is $G_1 \wedge G_2$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_1, G_2\}$, or
  (c) G is $G_1 \vee G_2$ and, for $i = 1$ or $i = 2$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_i\}$, or
  (d) G is $\exists x\, P$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{((\lambda x\, P)\ Y)\}$ where Y is a new (free) variable.

(2) *(Solving flexible goals)* $G \in \mathcal{G}_1$ has the variable Y of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$ as its head, and $\theta_2 = \{\langle Y, \lambda x_1 \ldots \lambda x_n\, \top \rangle\}$, $\mathcal{G}_2 = \theta_2(\mathcal{G}_1 - \{G\})$ and $\mathcal{D}_2 = SIMPL(\theta_2(\mathcal{D}_1))$.

(3) (*Backchaining step*) $\theta_2 = \emptyset$ and, for some rigid atom $G \in \mathcal{G}_1$ either

  (a) A is an atom obtained by instantiating the universal quantifiers in a clause in $\mathcal{P}$ with new variables and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$ and $\mathcal{D}_2 = SIMPL(\mathcal{D}_1 \cup \{\langle G, A \rangle\})$, or
  (b) $G' \supset A$ is obtained by instantiating the universal quantifiers in a clause in $\mathcal{P}$ with new variables and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G'\}$, and $\mathcal{D}_2 = SIMPL(\mathcal{D}_1 \cup \{\langle G, A \rangle\})$.

(4) *(Unification step)* For some flexible-rigid pair $\chi \in \mathcal{D}_1$, either $MATCH(\chi) = \emptyset$ and $\mathcal{D}_2 = \mathbf{F}$, or $\theta_2 \in MATCH(\chi)$ and $\mathcal{G}_2 = \theta_2(\mathcal{G}_1)$ and $\mathcal{D}_2 = SIMPL(\theta_2(\mathcal{D}_1))$.

A sequence of the form $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i \rangle_{1 \leq i \leq n}$ is a *$\mathcal{P}$-derivation sequence* for a goal formula G if (i) $\mathcal{G}_1 = \{G\}$, $\mathcal{D}_1 = \emptyset$ and $\theta_1 = \emptyset$, and (ii) for $1 \leq j < n$, the $(j+1)^{th}$ tuple is $\mathcal{P}$-derivable from the j^{th} tuple. Such a sequence terminates in failure if $\mathcal{D}_n = \mathbf{F}$ and with success if $\mathcal{G}_n = \emptyset$ and $\mathcal{D}_n$ is either empty of contains only flexible-flexible pairs. In the latter case, we say that the sequence is a $\mathcal{P}$-derivation for G. Such a sequence embodies in it a solution to the query G in the context of the program $\mathcal{P}$. The *answer* provided by this solution is obtained by composing $\theta_n \circ \cdots \circ \theta_1$ with any unifier for $\mathcal{D}_n$ and restricting the resulting substitution to the free variables of G.

An abstract interpreter for our language may be thought of as a procedure that, given a program $\mathcal{P}$, attempts to construct a $\mathcal{P}$-derivation for goal formulas. A state for this procedure is characterized by a set of goals and a disagreement set and the search for a solution is progressed by applying a simple step to one of these components. The need for ingenuity in picking a substitution does not exist in this procedure. However choices have still to be made between several alternative steps. In an actual implementation a depth-first search with the possibility of backtracking will be used to obtain a deterministic version of this procedure. In picking from the possible steps we always attempt to solve the problem posed by the disagreement set first, *i.e.*, we shall apply a unification step wherever possible. Several alternatives may exist even within this step and mechanisms for remembering these alternatives are needed. We consider these mechanisms and other relevant aspects in the following sections.

5 A Representation for Lambda Terms

We discuss now the machine representation of our typed lambda terms. The choice of this representation must, of course, be determined by the logical operations that need to be performed on these terms. We examine these operations below and develop, in an incremental fashion, the representation used within our implementation.

5.1 Renaming Bound Variables and the de Bruijn Notation

There is a need within $SIMPL$ to determine whether the heads of two rigid terms are identical. In carrying out this test, it may be necessary to consider a renaming of bound variables. For example, assume that we desire to unify the terms $\lambda y_1 \ldots \lambda y_n \, (y_i \; t_1 \; \ldots \; t_m)$ and $\lambda z_1 \ldots \lambda z_n \, (z_i \; s_1 \; \ldots \; s_m)$. These two terms are rigid and $SIMPL$ must reduce the task of unifying them to that of unifying the set

$$\{\langle \lambda y_1 \ldots \lambda y_n \, t_1, \lambda z_1 \ldots \lambda z_n \, s_1 \rangle, \ldots, \langle \lambda y_1 \ldots \lambda y_n \, t_m, \lambda z_1 \ldots \lambda z_n \, s_m \rangle\}.$$

However, prior to affecting this transformation $SIMPL$ must determine that the heads of the two terms in fact match. The case being considered indicates that, if the usual notation for lambda terms is employed, then this test involves not only a check for identity, but also the application of some α-conversion steps.

The frequency with which $SIMPL$ needs to be applied to terms makes it desirable to keep the check for compatibility of the heads of rigid terms as simple as possible. Thus, a representation of terms that eliminates the need for renaming steps appears preferable to the usual representation. Fortunately, a scheme that is suitable from this perspective has been devised by de Bruijn [6]. The essential idea used in this scheme is that the names of variables can be eliminated altogether by the use of indices. In the case of a bound variable occurrence, this index is obtained by counting the number of abstractions in a parse structure of the term up to the one binding the occurrence. As an example, the term $\lambda x \, ((\lambda y \, \lambda z \, (y \; x)) \; (\lambda w \, x))$ would be represented now by the expression $\lambda \, ((\lambda \, \lambda \, (\#2 \; \#3)) \; (\lambda \, \#2))$, where $\#i$ is the representation of index i. A term may also contain variable occurrences that are free in a global context. Some method for representing such variable occurrences is required that permits different occurrences of the same variable to be identified. The method employed in [6] for this purpose uses a fixed listing of all the variable symbols in conjunction with the term being represented. An index is then calculated for a free variable occurrence by counting the number of abstractions within which the occurrence is embedded and adding this to the position of the variable within the assumed listing. While this representation has its uses, we adopt a different one in our implementation: we represent free variables by cells in memory that are tagged as unbound variables. All occurrences of the same free variable correspond to (references to) the same cell and, thus, the requirement mentioned above is met by our representation. Free variables in our context are actually ones for which substitutions will be determined within the computational model described in the previous section. The representation chosen for these variables thus corresponds to the one usually employed within a WAM-like implementation, with the required cells being allocated on the stack or the heap. Note that there is an auxiliary virtue to our representation in that it makes it easy to perform substitutions into terms.

The de Bruijn representation solves the problem mentioned at the outset. To be specific, the two terms considered at the beginning are translated under this scheme into the de Bruijn terms $\lambda \ldots \lambda \, (\#i \; \hat{t}_1 \; \ldots \; \hat{t}_m)$ and $\lambda \ldots \lambda \, (\#i \; \hat{s}_1 \; \ldots \; \hat{s}_m)$, where, for $1 \leq i \leq m$, $\hat{t}_i$ and $\hat{s}_i$ are the de Bruijn representations of t_i and s_i respectively. Note that the heads of the two terms are identical under this representation. In general, the check for compatibility of the heads of two terms that must be performed by $SIMPL$ becomes an identity test and α-conversion need never be considered. There is also another benefit to the use of the de Bruijn notation: the abstractions that appear at the front of a term can be dispensed with in several situations. Such abstractions are needed within the unification process largely to provide a "context" within which to view two terms that are to be unified. When these contexts are identical, as would be the case within the de Bruijn scheme, they can be left implicit. To understand the pragmatic effect of this observation, consider the processing needed in conjunction with the two terms presented above. The task of unifying these two terms can be reduced to that of unifying the set $\{\langle \hat{t}_1, \hat{s}_1 \rangle, \ldots, \langle \hat{t}_m, \hat{s}_m \rangle\}$. Notice that the outer abstractions no longer need to be appended to the front of the argument terms. The operation to be performed by $SIMPL$ thus takes a very simple form: the task of unifying two rigid terms whose heads are identical reduces to that of unifying their arguments (of course, within an implicit context). This transformation can obviously be implemented much more easily than the earlier transformation that involved replicating the outside abstractions and adding these to the arguments.

5.2 Lazy Substitutions and the Suspension Notation

While handling α-conversion efficiently is important, the efficiency of an implementation of the higher-order language depends critically on the manner in which the operation of β-reduction is realized. At the logical level, the description of this operation requires some care in order to avoid problems of inadvertent capture of free variables. When the de Bruijn notation is used, this translates into a need to adjust indices in the course of performing this operation. For example, let us consider the term $\lambda x\, ((\lambda y\, \lambda z\, (y\ x))\ (\lambda w\, x))$ whose de Bruijn representation, as we noted, is $\lambda\, ((\lambda\, \lambda\, (\#2\ \#3))\ (\lambda\, \#2))$. This term reduces to the term $\lambda x\, \lambda z\, ((\lambda w\, x)\ x)$, whose de Bruijn representation is $\lambda\, \lambda\, ((\lambda\, \#3)\ \#2)$. Comparing the two de Bruijn terms, we notice the following: When substituting the term $(\lambda\, \#2)$ inside an abstraction, the index of the free variable, *i.e.*, 2, has to be incremented by 1. Further, indices in the scope of the abstraction that disappears on account of a β-contraction may have to be changed; here the index 3 corresponding to the variable occurrence x in the scope of the abstraction that is reduced must be decremented by 1. These general observations underlie the characterization of a substitution operation and the formalization of β-contraction based on this operation contained in [6].

A desirable feature in implementing the substitution operation inherent both in contracting β-redexes and in adjusting indices is the ability to perform this operation in a lazy fashion. Several advantages can be gained from this ability. For example, consider the task of determining whether the (de Bruijn) terms $((\lambda\, \lambda\, \lambda\, ((\#3\ \#2)\ s))\ (\lambda\, \#1))$ and $((\lambda\, \lambda\, \lambda\, ((\#3\ \#1)\ t))\ (\lambda\ \#1))$ can be unified. We assume that s and t denote arbitrary terms here. We can conclude that these two terms cannot be unified by observing that they reduce respectively to $(\lambda\, \lambda\, (\#2\ s'))$ and $(\lambda\, \lambda\, (\#1\ t'))$, where s' and t' are terms that result from s and t by appropriate substitutions. Note that in determining this fact it is not necessary to perform the potentially costly substitution operation on the terms s and t explicitly. However, to achieve this saving in effort we need to be able to represent some terms as combinations of other terms and substitutions; in the current context, we need a means for representing s' and t' as combinations of s and t with relevant substitutions. To see another benefit of lazy substitution, consider the reduction of a term of the form $((\lambda\, ((\lambda\, t_1)\ t_2))\ t_3)$ to head-normal form. Let t_2' be the term obtained from t_2 by substituting t_3 for the first free variable and decrementing the indices of all the other free variables by one. Then producing the head-normal form involves substituting t_2' and t_3 for the first and second free variables in t_1 and decrementing the indices of all other free variables by two. Each of these substitutions essentially involves a walk over the same structure, *i.e.*, the structure of t_1. It would obviously be beneficial if they could all be done in the same walk. However, the ability to combine the walks is dependent, once again, on the possibility of temporarily "suspending" a substitution generated by a β-contraction.

The idea of delaying substitutions is central to the implementations of functional programming languages. In these contexts, delayed substitutions are realized by the simple device of combining a term with an environment that represents bindings for the free variables that occur in the term. When the de Bruijn representation is used, this simple device is adequate only if the overall term is closed and if subterms embedded within abstractions need not be explored. These assumptions are acceptable within typical implementations of functional programming languages: only closed terms are considered in these languages and if a term is produced in the course of β-reduction that has an abstraction at the outermost level, then that term may be combined with its environment and returned as a *closure*. This idea is used, for instance, in the reduction machine described in [9]. Unfortunately, these assumptions are not appropriate in the context that is of interest to us. For example, the comparison of terms performed within $SIMPL$ requires the production of a head-normal form and this may require the contraction of β-redexes embedded within abstractions as well as the propagation of substitutions under abstractions. In these cases, a more complicated substitution operation needs to be encoded. For example, consider β-contracting a term of the form $((\lambda\, t)\ s)$ that appears embedded within some abstractions. Now, t might contain variables that are bound by outside abstractions. If the result of contracting this β-redex is to be encoded by the term t and an "environment", the environment must record not just the substitution of s for the first free variable in t but also the decrementing of the indices corresponding to all the other free variables. As another example, imagine that we wish to propagate an environment (or, alternatively, a substitution for free variables) under the abstraction in the term $(\lambda\, t)$. If the result is to be represented by a term of the form $(\lambda\, t')$ where t' is itself encoded as t and an environment, then this environment must be obtained from the earlier one by "shifting up" the index for the variables to be substituted for by one and adding an identity substitution for the variable with index 1. Further, the indices of the free variables in the terms that appear in the environment must themselves be incremented by one.

The problems described above can be overcome by using the notation developed in [33, 34, 35], and we use this notation in our implementation. The main novelty of this notation is the addition of a new category to terms that is called a *suspension*. In an informal sense, a suspension represents a "skeletal" term together with a suspended substitution. The precise syntax for lambda terms under the new notation is given by the following rules:

$$
\begin{aligned}
\langle term\rangle \quad &::= \quad \langle constant\rangle \mid \langle free\ variable\rangle \mid \#\langle positive\ integer\rangle \mid \\
&\qquad\ \lambda\,\langle term\rangle \mid (\langle term\rangle\ \langle term\rangle) \mid \langle suspension\rangle \\
\langle suspension\rangle \quad &::= \quad (\langle term\rangle, \langle index\rangle, \langle index\rangle, \langle env\rangle) \\
\langle env\rangle \quad &::= \quad nil \mid bndg(\langle term\rangle) :: \langle env\rangle \mid dum(\langle index\rangle) :: \langle env\rangle \\
\langle index\rangle \quad &::= \quad \langle natural\ number\rangle
\end{aligned}
$$

Expressions of the form (t, i, j, env) that are generated by these rules must satisfy certain additional constraints to be deemed well-formed: thinking of env as a sequence whose end is denoted by nil, (i) the length of env must be identical to i, (ii) for each element $dum(l)$ of env, it must be that $l < j$, and (iii) for each element of the form $bndg((t, k, l, e))$ in env it must be the case that $l \leq j$.

$((\lambda\, t), ol, nl, env) \longrightarrow \lambda\, (t, ol+1, nl+1, dum(nl) :: env)$

$((t_1\ t_2), ol, nl, env) \longrightarrow ((t_1, ol, nl, env)\ (t_2, ol, nl, env))$

$(cv, ol, nl, env) \longrightarrow cv$
provided cv is a constant or a free variable.

$(\#i, ol, nl, env) \longrightarrow \#j$
where $j = i - ol + nl$, provided $i > ol$.

$(\#i, ol, nl, env) \longrightarrow \#j$
provided $i < ol$, $env[i] = dum(nl_i)$ and $j = nl - nl_i$.

$(\#i, ol, nl, env) \longrightarrow (t_i, ol_i, nl, env_i)$
provided $i < ol$ and $env[i] = bndg((t_i, ol_i, nl_i, env_i))$.

$(\#i, ol, nl, env) \longrightarrow (t_i, 0, nl, nil)$
provided $i < ol$, $env[i] = bndg(t_i)$ and t_i is not a suspension.

Figure 2: Rules for Translating Suspension Terms

The meaning of the notation above can be explained by providing a translation from terms in this notation to de Bruijn terms. This is done through the set of rewrite rules that appear in Figure 2. In these rules, we use $env[i]$ to denote the ith element (from the left) in an environment. The rules presented apply to terms that are suspensions and their repeated application produces a de Bruijn term. The order of rule application can be arbitrary to an extent. To be precise, let us define the subterms of a term t as t itself and

(i) the subterms of t_1 and t_2 if t is of the form $(t_1\ t_2)$, and

(ii) the subterms of t_1 if t is of the form $(\lambda\, t_1)$ or of the form (t_1, i, j, env).

The de Bruijn term corresponding to a given term t is obtained by repeatedly applying the rules in Figure 2 to the subterms of t. The result produced is independent of the order of these rule applications[3]. These rules can actually be used in an incremental fashion and this is what is done in our implementation. When used in this mode, these rules produce the effect of performing substitutions lazily and they also permit the propagation of substitutions under abstractions.

The need to perform substitutions arises of course from β-contractions. Within our notation, this is reflected in a β-contraction rule that creates a suspension: under this rule, a term of the form

[3] To use a different terminology, these rules are confluent in this limited sense. They are not confluent in a broader sense in that the replacement of terms in env in a term of the form (t, i, j, env) is not permitted. This proviso can be relaxed in the case that the term in env does not have variables in it that are bound by outer abstractions, and this fact is made use of later in this section. We also note that a slightly different notation can be described together with operations that are confluent in the truest sense [34]. However, the present representation is preferable for certain reasons from the perspective of an actual implementation and is, therefore, the one we actually use.

$((\lambda\, t_1)\ t_2)$ gets replaced by the term $(t_1, 1, 0, bndg(t_2) :: nil)$. By using this rule in conjunction with the rules for propagating substitutions, some of the advantages of lazy substitutions mentioned earlier can be realized. However, one advantage that cannot be realized is the ability to combine the substitution walks generated by contracting multiple redexes. To facilitate this, an auxiliary β-contraction rule may be used. By this rule, a term of the form

$$((\lambda\, (t_1, ol+1, nl+1, dum(nl) :: env))\ t_2)$$

may be replaced by the term $(t_1, ol+1, nl, bndg(t_2) :: env)$. In a sense, this rule has built into it an operation of "merging" environments. While more sophisticated rules can be provided for combining environments, this rule suffices in several practical contexts and is also easily implementable. As an example of its use, let us consider the head normalization of the term $((\lambda\, ((\lambda\, t_1)\ t_2))\ t_3)$. As a first step in the reduction of this term to a head-normal form, we might produce the term $(((\lambda\, t_1)\ t_2), 1, 0, bndg(t_3) :: nil)$. The substitution may now be propagated so as to reveal a β-redex at the top level. This produces the term

$$((\lambda\, (t_1, 2, 1, dum(0) :: bndg(t_3) :: nil))\ (t_2, 1, 0, bndg(t_3) :: nil)).$$

At this point the second β-contraction rule is applicable and using it produces the term

$$(t_1, 2, 0, bndg((t_2, 1, 0, bndg(t_3) :: nil)) :: bndg(t_3) :: nil).$$

Notice that the substitutions generated by contracting the two β-redexes have been combined into one environment and can be performed in one walk over the structure of t_1.

We have, at this point, presented a notation for lambda terms and described some operations on this notation. To make this discussion coherent, several properties of our notation and operations must be examined. For example, there is a need to establish a correspondence between these and the usual notation and operations on lambda terms. Similarly, our various rewrite rules must be shown to be confluent if we wish to employ them in arbitrary order. Properties of this sort do hold as shown in [34, 35], and the various schemes and examples that we present here have a sound theoretical basis. We also note that there are several similarities between the notation we describe and those presented in [1] and [14]. We refrain from a discussion of these matters here, given that our focus is largely on the use of this notation in an actual implementation of our higher-order language.

5.3 Internal Representation and Implementation of Reduction

In using our notation in an actual implementation an important issue to consider is the internal representation of terms. The most natural such representation involves the use of a tagged cell with appropriate fields for the various syntactic categories. For constants and bound variables, the cell itself can hold all the relevant information — (a pointer to) the "name" of the constant or the index. In the case of an unbound free variable the content of the cell is unimportant and a binding can be recorded by converting this cell into a reference to another term. An abstraction cell must contain a pointer to the body of the abstraction, and an application cell must represent pointers to the two relevant terms. Finally, a suspension term requires the maintenance of its two indices, a pointer to the skeletal term and a pointer to the environment. An environment can be represented as a list and, in this form, admits of considerable sharing. For example, consider the term

$$(t_1, 2, 0, bndg((t_2, 1, 0, bndg(t_3) :: nil)) :: bndg(t_3) :: nil)$$

that arose in the context of the β-reduction discussed in the previous subsection. The final part of the environment of the overall term given by $bndg(t_3) :: nil$ and the environment of the term $(t_2, 1, 0, bndg(t_3) :: nil)$ can be represented by the same list. The rewrite rules used to produce the reduced term can be implemented to realize such sharing in a relatively direct manner.

Another issue that is of interest is that of the order in which the various operations on terms are to be performed. The main requirement is that of β-reducing terms to their head-normal forms at relevant points in computations. In general, it is necessary to explicitly expose only the heads of the reduced terms, and the arguments can be left as suspensions. Accordingly we adopt the strategy of reducing the leftmost, outermost β-redex at each stage, propagating substitutions only to the extent necessary to expose a β-redex or to reveal an atomic head. This strategy was, in fact, implicit in the reductions outlined earlier in this section. There are several advantages to this strategy some of which are obvious from other discussions in this section while others need detailed consideration. We do not examine these matters here, but only note that our strategy is a very simple one and, with respect to the contraction of β-redexes, is the one usually employed for reducing terms to their head-normal forms.

At a level of detail, it is of some importance to consider whether the various operations on terms should be implemented in a destructive or a non-destructive manner. To understand the tradeoffs involved, let us consider the reduction of the term $(\lambda\, (((\lambda\, t_3)\ t_2)\ t_1))$; we assume that t_1, t_2 and t_3 are arbitrary terms here. Based on the concrete representation just discussed, the internal structure of this term can be depicted by the graph in the left half of Figure 3; nodes in this figure correspond to cells and the labels are the tags on these cells. This term has a β-redex, given by the subterm $((\lambda\, t_3)\ t_2)$, that has to be β-contracted in producing a head-normal form. If the β-contraction is done non-destructively, the subterm would be left intact and a new subterm of the form $(t_3, 1, 0, bndg(t_2) :: nil)$ would be produced. Notice, however, that obtaining the overall term after this β-contraction involves copying the part of the term that appears above the β-redex. Thus, a non-destructive implementation of the β-contraction operation would eventually have to produce the structure shown in the right half of Figure 3. A destructive implementation, on the other hand, would replace the term $((\lambda\, t_3)\ t_2)$ in-place by its contracted form[4]. The effect of such an in-place change is felt directly in any term that the β-redex is a subterm of, thus obviating the additional copying.

The above discussion shows that a destructive implementation of β-contraction and other operations can yield savings both in space and time. A non-destructive realization actually has another associated problem. Consider, for instance, an attempt to head-normalize a term that is already in head-normal form. A naive implementation might replicate the structure of the term even in this case. However, this is undesirable: a mere "look-up" should not cause a new structure to be created. This problem can be overcome, but extra checks need to be incorporated into the normalization procedure, leading to an additional time overhead. Clearly, a similar problem does not arise with a destructive implementation.

While a destructive implementation appears preferable on several counts, there is one drawback associated with in-place changes in a logic programming setting that must be borne in mind: such changes need, in general, to be trailed to facilitate the resetting of a state upon backtracking. As

[4]Cells of different kinds must represent varying amounts of data, and so may be chosen to have varying sizes in a specific implementation. Achieving the effect of a destructive change may then require a "reference" cell to be used: the application cell corresponding to the β-redex may be changed into a reference to a (newly created) suspension term. Our comments here and below are independent of this possibility.

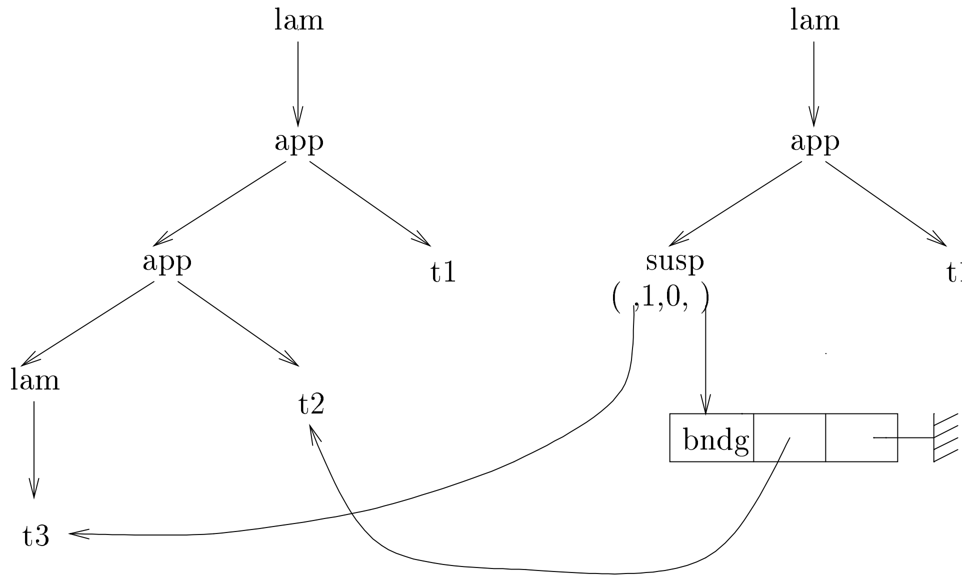


Figure 3: Non-destructive Reduction of Lambda Terms

a specific example, a β-redex may have resulted from a substitution for a variable. Undoing the substitution might thus have to be accompanied by the undoing of a reduction. In order to permit a complete resetting of an earlier state, the trailing operation must in the current situation save not only the address of the cell affected by a destructive change, but also the previous contents of cell.

Despite the mentioned drawback, we believe a destructive implementation is still the preferred one and that the benefits from using it will be substantial in practical situations. There are additional benefits, such as the sharing of reductions that we discuss below, that accrue from the use of this scheme. We therefore use this approach in our implementation. This choice indirectly determines another one that is pertinent to the implementation of logic programming languages. The destructive approach can only be adopted within the framework of a structure copying implementation. Thus, our machine model is driven closer at a level of detail to that of the WAM.

5.4 A Dependence Annotation on Terms

There is a refinement to our notation for lambda terms that has several practical benefits. This refinement consists of annotating terms to determine whether or not they contain variables bound by external abstractions. At the lowest level, we may distinguish bound variables as belonging to the first category and free (logic) variables and constants as belonging to the second. For abstractions and applications this categorization is determined in an obvious way by their subterms. With regard to a suspension, this determination depends on the category of the β-redex whose contraction yields it. Thus, consider the term $((\lambda\,(\lambda\,(\#1\ \#2)))\ c)$, where c is a constant. This term does not depend on external abstractions. The suspension that is produced by β-contracting this term, *i.e.*, the term

$$(\lambda\,(\#1\ \#2), 1, 0, bndg(c) :: nil)$$

must therefore not be so dependent and can be annotated as such. On the other hand the annotation on the suspension that results from β-contracting the term $((\lambda\,(\#1\ \#2))\ c)$ must reflect the fact that this term is dependent on an external abstraction.

There are two advantages to maintaining such an annotation with terms. One advantage is that in some cases the task of determining the underlying de Bruijn term corresponding to a given term can be simplified. Consider for example a term of the form (t, i, j, env) where it is known that t is a term that is not dependent on outside abstractions. Then this term can be simplified to (a pointer to) t. As another example, imagine trying to "read" a term of the form $(\#i, j, k, env)$ where $i < j$ and the ith element of env is of the form $bndg(t)$. Now if t is known not to be dependent on outside abstractions, the suspension term can be simplified once again to a pointer to t. In both cases, a fair amount of work can be avoided as a result of the suggested annotation. An auxiliary benefit is that the simplification fosters a sharing of terms that can in turn lead to a later sharing of reduction steps. For example, consider again the term (t, i, j, env) but assume now that t is of the form $(t_1\ t_2)$ and is actually a β-redex. Further, assume that t is "shared" with other terms in the computational context. Now in attempting to reduce the suspension term to a head-normal form in a situation where no annotations are maintained, we would first produce the term $((t_1, i, j, env)\ (t_2, i, j, env))$. This term is eventually identical to the earlier term $(t_1\ t_2)$ and will have to be β-contracted in producing a head-normal form. Notice, however, that the "reading" step produces a new application term and thus breaks the sharing that initially exists with respect to this β-redex. The contraction of the β-redex that is obtained from $((t_1, i, j, env)\ (t_2, i, j, env))$ will thus have no effect on other occurrences of the β-redex $(t_1\ t_2)$. In contrast if the term (t, i, j, env) is first simplified to a pointer to t, the sharing of the β-redex is maintained and its contraction will be propagated to all the other places where it is needed.

A similar observation concerning sharing applies to terms appearing in the environment; in fact other forms of sharing are fostered first through sharing in the environment. We present a concrete illustration of the benefits of the suggested annotation scheme in Figure 4 by considering the head-normalization of a term that in the usual notation might be written as

$$((\lambda x\,(x\ (c\ x)))\ ((\lambda y\,(Z\ b))\ a));$$

the symbols a, b and c are assumed to be constants in this expression. The syntax that is used for terms in this figure is obtained from the rules presented earlier by replacing the one for the category of terms by the following:

$$\begin{array}{rcl} \langle term\rangle & ::= & var() \mid ref(\langle term\rangle) \mid \\ & & const(< Constant\ Name >\rangle) \mid bv(\langle positive\ integer\rangle) \mid \\ & & clam(\langle term\rangle) \mid flam(\langle term\rangle) \mid \\ & & capp(\langle term\rangle, \langle term\rangle) \mid fapp(\langle term\rangle, \langle term\rangle) \mid \\ & & csusp(\langle term\rangle, \langle index\rangle, \langle index\rangle, \langle env\rangle) \mid \\ & & fsusp(\langle term\rangle, \langle index\rangle, \langle index\rangle, \langle env\rangle). \end{array}$$

The syntax described here reflects the internal representation discussed in the previous subsection: the functor in each case denotes the tag on the corresponding cell and the arguments denote the information maintained in the cell. The nomenclature for the tags is self-explanatory. We note only that the tag *clam* is used for an abstraction that is independent of external abstractions and

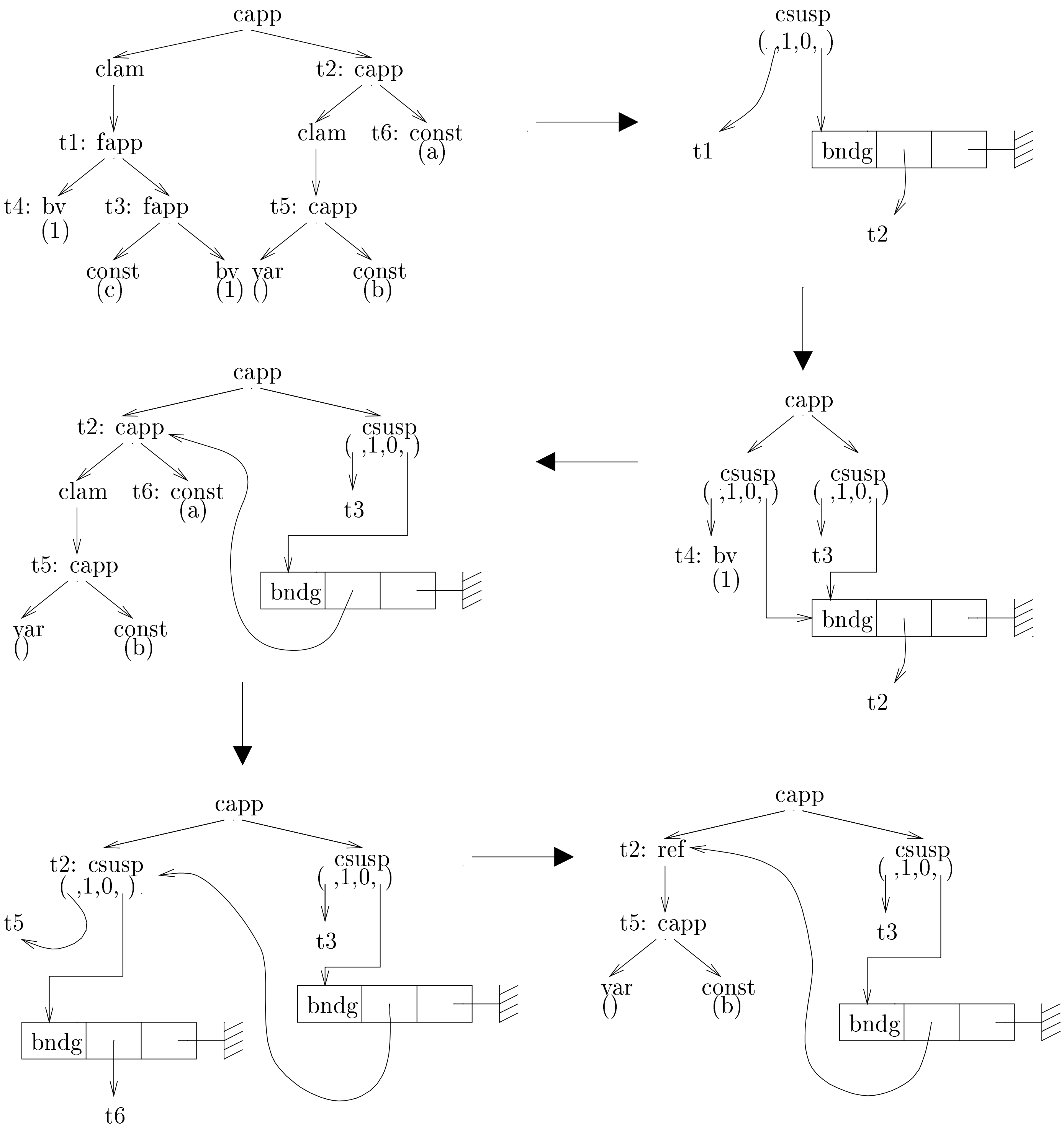


Figure 4: Head Normalization of the Term $((\lambda x\,(x\ (c\ x)))\ ((\lambda y\,(Z\ b))\ a))$

flam is used otherwise, and that a similar distinction is to be made in the cases of applications and suspensions. To simplify the presentation of our example, we have labelled some of the subterms appearing in it and have indicated the "destination" of pointers by using these labels. The association between a term and a label L is indicated in Figure 4 by prefacing the root of the term with "`L:`". These labels in fact correspond to the location of the cell used to store (the root node of) the term.

Given the obvious advantages to our annotation scheme, we adopt this in our implementation. We note that a similar suggestion is also made in [4] in the context of the conventional notation for terms and that some of the benefits discussed here are also obtained in that context.

5.5 Representing Types

In the discussion up to this point we have not considered explicitly the fact that the lambda terms of interest to us have types associated with them. These types have a twofold role in the language [32]. At one level, they serve to limit the set of acceptable programs. At another level, they participate in the computational mechanism of the language; this role is apparent from the manner in which types determine the imitation and projection substitutions that are to be generated for a flexible-rigid pair. The first function of types is pertinent to the process of compilation but is not relevant to the execution of a program. Thus, this use of types does not have a bearing on the internal representation of terms. As for the second purpose, we observe that it is sufficient to maintain types with only the constants and free variables appearing in lambda terms. Maintaining such annotations is also necessary: the types of free variables are needed for both the imitation and projection substitutions and the types of constants are needed in determining the imitation substitutions. Although the types of bound variables are not strictly needed, we maintain these as well in our implementation. The advantage of doing this is that the type of any lambda term in a program can then be completely reconstructed since this is fixed by the types of the atomic symbols that appear in the term. Thus, there is the capability within our implementation of displaying the types of terms, should this be desired in conjunction with computed answers.

The representation of types adds a extra component, *i.e.*, a pointer to a type, to the cells for atomic symbols[5]. Although this is not presented explicitly here, a concrete structure for the various cells used in our implementation can be visualized based on the discussions in this section.

6 Run-time Structures for Higher-Order Unification

The next matter of concern is the special machinery that should be included in our implementation for dealing with higher-order unification. Based on the abstract interpreter presented in Section 4, we observe three new aspects present in the processing model for our language that require additional consideration. First, one of the components characterizing a state in our abstract interpreter is a disagreement set and some method for representing these must be included in an actual implementation. It might appear that an explicit representation of such sets may be avoided as in the first-order case. However, this is not a reasonable possibility for the higher-order language, as we

[5]In the presence of polymorphism, this pointer is replaced by two: a pointer to a type "skeleton" and a pointer to a type environment. This representation of types and its use is discussed in the context of a first-order language in [21].

argue below. Second, unification has a branching character in the higher-order context and mechanisms are needed for dealing with this. The usual approach adopted within logic programming languages in the face of choices is to conduct a depth-first search and we follow this course here as well. From the perspective of an implementation, this calls for the encoding of the choice points in unification in the state of the machine and a method must be provided for doing so. Finally, the comparison of terms within unification is based on the use of head normal forms. Terms are not necessarily in normal form at the points where they have to be compared. An implementation must therefore include machinery for doing the necessary conversion during execution. We examine each of these aspects in detail below and propose additions to the usual mechanisms of the WAM for dealing with each of them.

6.1 Representing Disagreement Sets

Disagreement sets arise in principle even in the context of first-order unification. However, the decidability of this unification and the existence of most general unifiers for first-order terms permit the task of unifying these terms to be recursively decomposed into that of unifying their respective subterms. An explicit representation of disagreement sets can therefore be avoided: the recursive algorithm can be implemented as in the WAM through the sequential execution of compiled code or, for that part of unification done "interpretively," through the use of a pushdown list.

Neither of the properties mentioned for first-order unification holds in the higher-order context. Nevertheless, it is still possible in principle to adopt a recursive procedure that attempts to unify subterms in order and that explores the suitability of unifiers for the subterms in a depth-first fashion. However, basing the unification process entirely on this approach appears undesirable for pragmatic reasons: in a situation where choices have to be made in substitutions, it appears best to use all available constraints on these choices. For example, let us assume that we desire to unify two terms of the form $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$, where f is a constant (function) symbol and, for $1 \leq i \leq n$, t_i and s_i are arbitrary terms. Following the approach being considered, we may attempt to do this by unifying the pairs $\langle t_1, s_1 \rangle, \ldots, \langle t_n, s_n \rangle$ in sequence. In the course of unifying the pair $\langle t_1, s_1 \rangle$, it may be necessary to choose a substitution for a variable x. This variable may appear in other pairs of subterms as well and several of the choices of substitution for x may render these pairs non-unifiable. Using this information may, at the very least, curtail the branching in search. In specific situations this may even make a difference between finding and not finding a unifier: a particular choice of substitution for x that is eliminated by other considerations may lead to a neverending search for a unifier for $\langle t_1, s_1 \rangle$.

The above discussion suggests the following approach for finding unifiers for a disagreement set. At any point in the computation, we pick a pair from the set and proceed to search for a unifier for only this pair till such time that several possibilities arise in furthering the search. At such points, we examine the effect of a possible substitution on the rest of the disagreement set before proceeding further. Implementing this approach clearly requires an explicit representation and manipulation of disagreement sets within the unification process. A further consideration of the issue reveals situations in which disagreement sets may have to be carried even between invocations to the unification procedure. The unification step of the abstract interpreter described in Section 4 is applicable to a state only if the associated disagreement set contains a flexible-rigid pair. However, a disagreement set might be generated that is nonempty but contains only flexible-flexible pairs. For reasons observed already, it is best to "suspend" the unification process on encountering such a disagreement set, and our abstract interpreter implicitly manifests this behavior. The application of a backchaining step may introduce further constraints and eventually cause the suspended unification problem to be reactivated. Within this general scheme, a disagreement set is an explicit part of a state of our machine and an encoding must therefore be provided for such sets.

An illustration of the preceding remarks is perhaps useful. For this purpose, let us consider the $mapfun$ predicate from Section 3 whose definition is reproduced below.

$(mapfun\ []\ F\ [])$.
$(mapfun\ [X|L1]\ F\ [(F\ X)|L2])$:- $(mapfun\ L1\ F\ L2)$.

Assume now that we are interested in solving the query

?- $(mapfun\ [a,b]\ G\ [(G\ b),(h\ a\ b)])$.

The first step taken by our abstract interpreter would be to try to unify this query with the heads of the available clauses. As we observe in Section 8, much of this step can be compiled, and the query itself can be transformed into an invocation of the compiled code for $mapfun$. Regardless of the particular implementation, we see that the attempt to unify the query with the head of the first clause will fail. Attempting to use the second clause gives rise to the disagreement set

$$\{\langle (mapfun\ [X|L1]\ F\ [(F\ X)|L2]), (mapfun\ [a,b]\ G\ [(G\ b),(h\ a\ b)\rangle\}$$

that is eventually simplified to

$$\{\langle X,a\rangle, \langle L1,[b]\rangle, \langle F,G\rangle, \langle (F\ X),(G\ b)\rangle, \langle L2,[(h\ a\ b)]\rangle\}.$$

Most of the pairs in this set can be solved by rather straightforward substitutions. Thus, substituting a for X, $[b]$ for $L1$, G for F, and $[(h\ a\ b)]$ for $L2$, this set can be reduced to the set $\{\langle (G\ a),(G\ b)\rangle\}$. The only pair in this set is a flexible-flexible one and hence no further unification steps are applicable[6]. An attempt is now made to solve the goal $(mapfun\ [b]\ G\ [(h\ a\ b])$, *i.e.*, a suitably instantiated version of the body of the second clause. As the reader can ascertain, our interpreter solves this goal by proposing two different substitutions for G: $\lambda x\,(h\ a\ x)$ and $\lambda x\,(h\ a\ b)$. Instantiating G with either of these substitutions changes the status of the suspended disagreement pair and it actually provides a "filter," rejecting the first solution but accepting the second.

Having noted that disagreement sets need to be represented explicitly, we turn now to the choice of representation for these. At least the following factors should be considered in making this choice:

(i) Disagreement sets change in an incremental fashion. Typically substitutions that are made affect only some of the pairs leaving others unchanged. Thus, in representing a newly generated disagreement set it would be best if the unchanged portions of the old set were reused, *i.e.*, a sharing of structure between sets should be permitted.

[6]In this case a solution can be provided for the pair that has as instances all other solutions, namely the substitution $\lambda x\,Y$ where Y is a (free) variable distinct from x. Nevertheless, there is no systematic method for generating such substitutions in the general case and in any case this substitution will not be considered by the abstract interpreter described in Section 4.

(ii) It should be possible to rapidly reinstate disagreement sets that were in existence earlier. This facility is needed because our computational model permits backtracking and is especially pertinent if our representation permits structure sharing of the kind mentioned in (i) above.

The representation employed in our implementation is based on a stack of disagreement pairs that we call the disagreement set or DS stack. This stack is empty at the beginning and new pairs are added to the top of it as computation proceeds. Space from the stack is reclaimed only when backtracking occurs; thus, this stack has much of the character of the heap in the WAM. The set of disagreement pairs active at any point consists of only some of the pairs in this stack and is given by a *live list* that threads together the relevant pairs. To facilitate the reinstatement of disagreement sets active earlier, this list is actually maintained as a doubly-linked one. Conceptually, there are two ways in which an active set might be affected by computation. First, there might be a call to $SIMPL$; this might occur explicitly after a substitution is determined or implicitly through compiled code as explained in Section 8. Such a call may require some of the elements in the live list to be removed and new ones to be added. The former operation is performed by splicing the relevant elements out of the live list and putting pointers to them on a separate trail stack that is called the *DS trail*. The addition of new elements is effected by first placing them on the top of the DS stack and then adding them onto the end of the live list. The second way in which the active set is affected is by a backtracking operation. In this mode, the changes caused by $SIMPL$ have to be undone. This is effected by reinserting the trailed "dead" elements into the live list and by discarding the new live elements added at the top of the DS stack.

Our representation of disagreement sets clearly meets the requirements mentioned earlier. The realization of the overall scheme calls for certain additions to the WAM machinery. First of all, we need the new DS stack and the DS trail. In addition we need a set of new global registers: the DS and DS_TR registers that indicate the tops of the DS stack and DS trail and the LL and LLE registers that point to the beginning and the end of the live list. Finally, we need to maintain a record of the new registers at the points to which backtracking can take place. This requires the addition of four new fields to the WAM choice point records and the inclusion of these fields in branch point records, a new kind of record catering to branching in unification as explained below.

It is in principle possible to perform all the processing needed within $SIMPL$ using only the DS stack. However, any addition to this stack remains there until backtracking takes place. Given this continually growing nature of the DS stack, judiciousness should be exercised in making additions to it. With this requirement in mind, we observe that when a rigid-rigid pair is encountered within $SIMPL$, the process of simplification can be applied recursively to the subterms and additions to the DS stack (and live list) need not take place till a flexible-flexible or a flexible-rigid pair is encountered. Using this idea in the $mapfun$ example considered earlier in this section, for instance, it necessary to contemplate an addition to the DS stack of only those pairs that appear in the set

$$\{\langle X, a\rangle, \langle L1, [b]\rangle, \langle F, G\rangle, \langle (F\ X), (G\ b)\rangle, \langle L2, [(h\ a\ b)]\rangle\}.$$

In implementing this idea, we take advantage of a pushdown list as in the WAM. In this process, we make use of the similarity noted earlier between $SIMPL$ and the simplification operation that is used in first-order unification.

There are certain situations in which a most general unifier can be readily generated for a given disagreement pair. A simple example of this arises from first-order unification: given a pair of the form $\langle X, t\rangle$ where X is a variable of atomic type and t is a term in which X does not appear, all unifiers for the pair must be instances of the unifier that substitutes t for X. Alternatively, if X does occur in t, failure in unification can be registered immediately. This observation can be generalized somewhat in the higher-order context: given a pair of the form $\langle \lambda x_1 \ldots \lambda x_n\, X, \lambda x_1 \ldots \lambda x_n\, t \rangle$, where X has an arbitrary type, X can be bound to t and this pair can be removed from the set provided neither X nor the variables in $\{x_1, \ldots x_n\}$ appear in t. (The use of the de Bruijn notation makes the binders identical and also simplifies the verification of the second condition). Failure does not result in the higher-order case if the conditions indicated are not satisfied. However, failure can still be registered in certain situations and in other situations the search for unifiers can be simplified by binding X to a term that mimics part of the structure of t and by adding additional pairs to the disagreement set. Notice that in some cases flexible-flexible pairs can be solved directly by this process; for example, the pair $\langle F, G \rangle$ that arose earlier in this section can be solved in this manner. Using observations such as these reduces the need to consider the general imitation and projection substitutions and hence also the attendant bookkeeping steps. In the $mapfun$ example, the disagreement set can in fact be reduced to $\{\langle (G\ a), (G\ b) \rangle\}$ by these means. Significantly, first-order unification can be solved deterministically using these observations and a similar set of observations suffice for a slightly richer unification problem [24]. Experimental evidence [22] indicates that a majority of the unification problems that arise in practice fall into one of the mentioned categories, suggesting the general importance of incorporating these observations into an implementation.

The appropriate place to consider such substitutions is within $SIMPL$. Doing this and also being conservative in the additions to the DS stack now calls for the use of two pushdown lists. The general scheme works as follows. The simplification of a disagreement set proceeds as before with the use of the first pushdown list, except that the process may now also involve making bindings to variables. When the process "bottoms out" with a flexible-flexible or flexible-rigid pair, this is pushed onto the top of the second pushdown list instead of the DS stack. When all the pairs in the original disagreement set are simplified, a check is made to see if a binding was made during the simplification process; a global variable called `BD` is set to true when a binding is made and the required test thus involves only checking the status of this register. If no binding was made, the pairs in the second pushdown list are added to the top of the DS stack and included in the live list. If a binding was made, the simplification phase is repeated with the disagreement set being given now by the live list *and* the pairs in the second pushdown list and the roles of the two pushdown lists reversed.

6.2 Branching in Unification

Alternatives in unification are encoded in a state in our machine by using a structure called a *branch point record.* These records are similar to the choice point records of the WAM and are also maintained on the stack. Branch point and choice point records form one linear sequence from the perspective of backtracking and are therefore linked together as such on the stack. As in the WAM, the `B` register points to the start of this sequence.

The specific information stored in a branch point record differs, of course, from that maintained in a choice point record. Branch point records are created when a choice is exercised in picking a substitution based on a flexible-rigid pair. As we have noted, the imitation and projection substitutions are completely determined by the heads of the flexible and rigid terms. Thus, from the perspective of finding the next substitution, it is necessary only to maintain these heads and to remember which substitutions have been tried. However, one piece of information is used repeatedly in generating the various substitutions and this is the target and argument types of the flexible head. It therefore appears worthwhile to determine these types once at the time of creating the branch-point and record them for use when constructing the substitutions. In remembering the substitutions that have been tried, we note the following: if the projection substitutions are attempted in sequence and the projection and imitation substitutions are attempted in a fixed order, it is sufficient to record only the "number" of the projection substitution tried last (or to be tried next). From practical experience, it appears worthwhile to let the user of the language determine the order in which the imitation and projection substitutions are tried and to even change this order in the course of execution. To allow for this, one extra piece of information needs to be recorded: the order being used at the time the branch point was created.

A branch point record accordingly contains the following fields for the purpose of constructing substitutions:

(i) Pointers to the heads of the rigid and flexible terms respectively.

(ii) A boolean field that is set to *true* if projection substitutions were being attempted first and to *false* otherwise, and an integer field, initialized to 0, that records the last projection substitution tried.

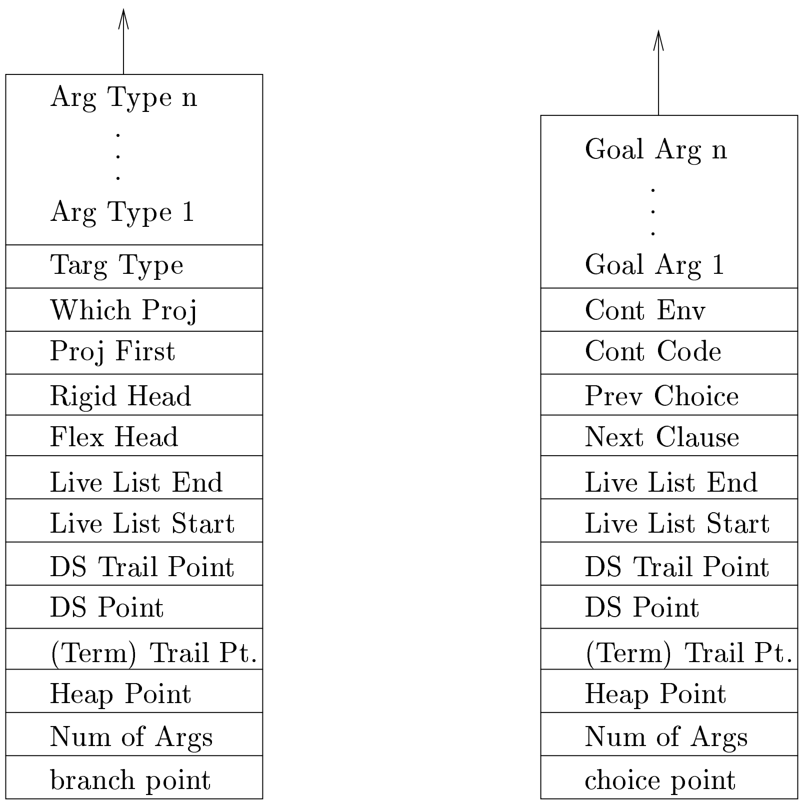
(iii) A number of fields that point successively to the target and argument types of the head of the flexible term.

(iv) A field that records the number of argument types for the head of the flexible term. This is needed for determining the size of the branch point and for knowing when all projection substitutions have been tried.

A branch point record must also maintain some information for the purpose of resetting state as a result of backtracking. As with a choice point, two fields are needed to note the top of the heap (obtained from the `H` register) and the top of the (term) trail (obtained from the `TR` register) at the time of creation of the branch point record. The representation of disagreement sets discussed earlier in this section necessitates two additional fields that have a similar function with respect to the DS stack and DS trail. Finally, two more fields are required for recording the beginning and end of the live list respectively. The last four fields must also be added to choice point records for an identical purpose in our implementation.

There are two final components to a branch point record. First, the previous choice or branch point record needs to be noted and a field is needed for this purpose. Second, the particular action to be performed on backtracking to a branch point differs somewhat from that to be carried out on returning to a choice point. It is therefore necessary to discriminate between these two kinds of records, and a suitable annotation is needed for this purpose. The last field of a branch point record is the requisite annotation, and a similar field must be added to a choice point record. The complete structure of a branch point record is presented in Figure 5, together with the (modified) structure of a choice point record.

Projection and imitation substitutions require new terms to be generated. These terms are best allocated on the heap since backtracking then permits the space to be reclaimed, and this is the course followed in our implementation.



(a) Branch-Point Record (b) Choice-Point Record (Modified)

Figure 5: Bookkeeping Devices for Backtracking

6.3 Normalizing Terms

The various operations used within unification depend on terms being presented in head-normal form. The terms that arise in the course of executing a program are often not in this form; for instance, consider a term after a binding such as that dictated by imitation or projection is made to a variable appearing in it. It is therefore necessary to consider normalization procedures in an implementation of our language as also the interaction between such procedures and the ones that need to use the results produced by them.

The head-normalization procedure that is used in our implementation is based on the discussions in Section 5. The notation for terms described therein permits a rather simple stack based implementation of such a procedure: the chief virtue of the notation in this regard is its encapsulation of the substitution arising out of a β-contraction as a truly atomic operation. As mentioned already, the strategy used in normalization is that of reducing the leftmost outermost redex at each stage. In particular, our reduction procedure looks at the term at the top of a reduction stack called the SL stack and a global register `NUMARGS` to determine its next step. A reference to the term to be reduced is initially placed on the top of the SL stack and the `NUMARGS` register is initialized to 0. The specific action taken by the procedure at each point is based on the rewrite rules presented in Section 5. If the term on the top of the SL stack is a suspension, a step is taken towards exposing its "true" top-level structure. This may involve the use of the SL stack. For example, if the "skeleton" term is itself a suspension, then its top-level structure must be exposed first, leading to a reference to it being placed on the top of the SL stack. If the term is an application, it is popped from the SL stack, references to its "operator" and "operand" parts pushed onto the SL stack and the `NUMARGS` register is incremented. If the term is an abstraction, the action taken depends on the contents of the `NUMARGS` register. If it is 0, the term is replaced by a reference to the body of the abstraction. If it is not 0, then a reference to the required argument lies just below it in the SL stack. The top two items on the SL stack are popped, the appropriate suspension term is created and a reference to it is pushed onto the top of the SL stack. The procedure eventually halts when the top of the SL stack contains a reference to an atomic term, *i.e.*, to a constant or a bound or free variable.

A detailed presentation of the procedure outlined above appears elsewhere [35] and we therefore do not undertake it here. However, the structure of this procedure is apparent from the description provide above and this suffices for discussing its interaction with other parts of our implementation. Assume that this procedure is invoked with a term that can be β-reduced to the form $\lambda x_1 \ldots \lambda x_n \, (h \; t_1 \; \ldots \; t_m)$, where h is a constant or variable. When the procedure is finished, the SL stack will contain, in consecutive locations from the top, references to the de Bruijn representation of h and the suspension representations of terms $s_1, \ldots, s_m$ that are β-convertible to $t_1, \ldots, t_m$[7]. This presentation of the "matrix" of the term is particularly convenient for the other operations required within unification. To begin with, the heads of terms and their status, whether rigid or flexible, is easily determined. At a more detailed level, assume that the simplification operation is to be applied to two terms t and r whose head-normal forms have identical binders. The head-normalization procedure can be invoked to produce the matrices of these two terms in different segments of the SL stack. Then, if the two terms are rigid and have identical heads, the terms out of which new disagreement pairs have to be formed appear at the same displacement from different starting locations, thereby facilitating an iterative structure to further processing. This observation is also made use of in the compilation of parts of $SIMPL$ considered in Section

[7]SL in the name of the reduction stack is an acronym for "structures lists," a choice justified by this observation.

8.

The presentation of $SIMPL$ assumes that the lengths of binders of two terms to be unified can be adjusted to be identical. In reality this is not a situation that holds automatically but, rather, one that must be achieved at the required points in computation by using the η-conversion rule. The normalization routine described here provides information relevant to making the requisite adjustment. In particular, assume that we are comparing two rigid terms of the form $\lambda x_1 \ldots \lambda x_n\,(c\ t_1\ \ldots\ t_l)$ and $\lambda x_1 \ldots \lambda x_m\,(c'\ s_1\ \ldots\ s_k)$. The values l and k are produced in the `NUMARGS` register when the respective terms are normalized. These values can be used to determine which term needs extra arguments and also the number of such arguments that are to be provided. Further, assuming the suspension notation, the specific adjustment can also be carried out easily. For instance, assume that $l > k$ and c' is identical to one of $x_1, \ldots, x_m$. Then the de Bruijn index for c' in the adjusted form is given by adding $(l - k)$ to the original index. Similarly, if s'_i is the term in suspension notation corresponding to s_i, the desired adjustment is encapsulated in the term $fsusp(s'_i, 0, j, nil)$ where $j = l - k$.

The normalization procedure used in our implementation requires the creation of new structures for terms at various points in its execution. These terms are once again best allocated on the heap and it is this course that is adopted in our implementation.

7 The Overall Structure of the Machine

Our machine integrates the various components discussed in the last two sections into the basic structure of the WAM. Processing within this machine is initiated by the attempt to solve a rigid atomic goal. This involves matching the goal with the head of a clause. The choice of clause can be limited as usual by using the "name" of the atomic goal. Within this set, the alternatives are tried in a depth-first fashion, and a choice point record is created to remember the possibilities that might have to be tried later. A new copy of the selected clause is obtained by allocating an environment record on the stack. Matching the atomic goal with the head of the clause now requires a unification problem to be solved. Attempting to solve this problem may call upon operations such as normalization of terms and simplification of disagreement pairs, and the various devices described in the previous section are used for this purpose. In several instances the unification problem can be solved deterministically and in its entirety by a simplification process and by making straightforward bindings; this may in fact be achieved through compiled code, as explained in the next section. However, in some cases truly higher-order substitutions may have to be considered. In such instances, a flexible-rigid pair is chosen, a branch point record is created, a projection or imitation substitution is picked and effected on the terms, and the modified unification problem is reconsidered. The unification problem may eventually be solved, perhaps by producing a disagreement set that contains some flexible-flexible pairs. Execution then continues by attempting to solve (a suitably instantiated copy of) the body of the clause or, in the case the body is empty and the atomic goal is but one component of a conjunctive one, some other component of the original goal.

There are places within this cycle where failure might occur. This results in a reconsideration of the last choice made. The relevant alternatives are obtained by examining the record pointed to by the `B` register. Regardless of what kind of record this might be, the first step is to recreate the state of the machine that existed prior to the current alternative having being picked. This process involves "unwinding" the term and DS trails and resetting the `LL` and `LLE` registers. The next step depends on the particular kind of record. In case this is a choice point record, the program pointer, *i.e.*, the `P` register, is set to the address of the code for the next alternative to be tried; this address is obtained from the record in question. (We assume a model based on compilation here, but the general idea applies to an interpretive model as well). If, on the other hand, the `B` register points to a branch point record, control transfers to code for picking the next substitution. All the information needed for determining this substitution is present in the relevant record. If all alternatives have been tried, the branch point record is discarded by resetting the `B` register to the preceding backtrack point indicated in the record, and backtracking is initiated again. Otherwise a substitution is picked, the relevant changes are registered in the branch point record and execution proceeds by reexamining the unification problem posed by the active disagreement set.

Figure 6 depicts the various components of our machine and provides a snapshot of its state in the course of the processing cycle outlined in this section. One perhaps intriguing aspect of this figure is that it indicates the `S` register as an index into the SL stack instead of the heap. This is a manifestation of the fact that the top-level structure of lambda terms may become apparent only after some normalization steps, and is explained in greater detail in the next section.

8 Compilation

Ideas similar to those employed in the WAM can be used to compile much of the action to be taken in solving a goal in our language, and we discuss this issue below. For the purpose of describing the compilation process, we categorize goals as rigid atomic goals, complex goals whose top-level logical structure is known statically and flexible atomic goals. The last kind of goal may be instantiated to a goal in one the earlier two categories at runtime. We discuss compilation aspects pertinent to each of these cases and then illustrate these ideas in the context of programs presented earlier earlier in the paper.

8.1 Compiling Rigid Atomic Goals

The attempt to solve a rigid atomic goal begins conceptually with the application of $SIMPL$ to this goal and the head of a clause. This attempt quickly fails if the "names" of the goal and the head of the selected clause are distinct. Several clauses can be summarily rejected on this basis, and the head of the goal can in fact be used to index into the code for the clauses that should be tried. Given the depth-first, backtracking nature of our search, these clauses must be attempted in sequence and choice points need to be set up to record the remaining alternatives. This is just as in the first-order context and the ideas and instructions employed in the WAM can be used to realize the desired effects. Under this scheme, the code for all the clauses defining a predicate with the same name is knit into one piece with special instructions for clause sequencing, establishing and maintaining choice points and permitting a further indexing based on the structure of specific arguments of the goal. We assume the reader is familiar with these ideas and we use the needed instructions below without further discussion. The uninitiated reader may consult [39] or [2] for details.

The simplification operation within the context of each clause reduces to matching each of the arguments of the goal with the corresponding argument of the clause head. The actual arguments can be made available in WAM fashion via argument registers. As noted already, the application of $SIMPL$ and the making of simple bindings in conjunction with each argument pair bears close

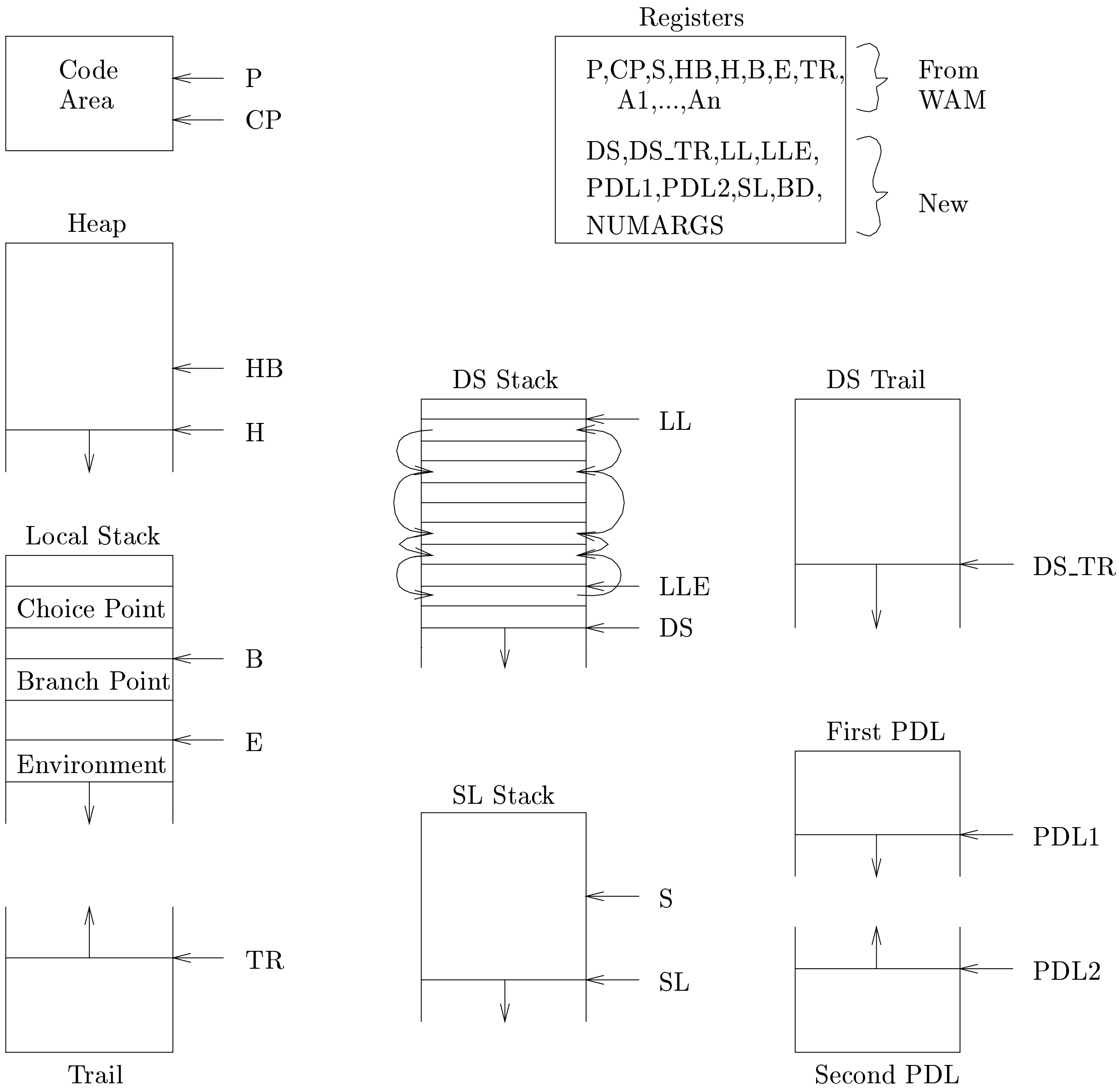


Figure 6: State of the Machine

similarity to the processing that must be done in the context of first-order terms. Thus, these actions can be compiled in a similar fashion, and our machine includes "unification" instructions akin to those in the WAM for this purpose. However, the specific instructions that we use are somewhat different from those in the WAM. These differences arise from our use of a richer class of terms, from the need to perform a normalization operation on these terms and from the difference in representations for our terms and for first-order terms. We discuss these aspects below.

The first factor mentioned above results in a richer instruction set. One category of instructions arises from the the need to create arbitrary lambda terms. The instructions introduced for this purpose consist of the following:

(i) `put_clambda Ai,Xj` and `put_flambda Xi,Xj`, used to construct abstractions. The effect of these instructions is to push a *clam* or *flam* cell onto the heap with contents obtained from `Xj` (which must be a reference to a term) and to make `Ai` or `Xi` a reference to the resulting cell.

(ii) `put_capp Ai,Xj,Xk` and `put_fapp Xi,Xj,Xk`, used for constructing applications. These instructions result in a *capp* or *fapp* cell being pushed onto the heap. The subterm pointers for these cells are set using `Xj` and `Xk` and a reference to the resulting cell is placed in `Ai` or `Xi`.

(iii) `put_index Xi,j,type`, used for constructing bound variables. This instruction results in a *bv* cell being pushed onto the heap. The index of this cell is set to j and the type component is set to `type` (which must be a reference to the type structure for the bound variable). Finally, a pointer to the created cell is placed in `Xi`.

As an illustration of the use of these instructions, the creation of the term $\lambda z\, \lambda y\, (g\ z\ y)$ would be compiled into the following sequence, assuming `t1` and `t2` are references to the argument types of g:

```
put_const g,X1
put_index X2,2,t1
put_fapp X1,X1,X2
put_index X2,1,t2
put_fapp X1,X1,X2
put_flambda X1,X1
put_clambda A1,X1
```

The `put_const` instruction used in this sequence is borrowed from the WAM and has an identical effect. At the end of this sequence, the desired term resides on the heap and a pointer to it is placed in the register `A1`.

The use of lambda terms also necessitates two additional instructions for unification. These instructions are `simpl Vi,Vj` and `finish_unify` and are required for the situations where unification cannot be performed entirely through compiled code. The first instruction performs the simplification function, augmented with the binding operation outlined in Section 6, on the pair formed by `Vi` and `Vj`. The pushdown lists are used in this process and disagreement pairs are added to the active disagreement set only when further simplification is not possible. This instruction behaves very much like the `get_value` instruction of the WAM when confronted with first-order (or first-order like) terms and is in fact a replacement for that instruction in our machine. The second instruction occurs as the last of the unification instructions generated for a clause head. It checks the status of the live list. If this is empty, as would be the case for first-order programs, it acts as a no-op. Otherwise it initiates the "interpretive" unification cycle described in the previous section.

Much of the first-order-like unification can be compiled using the `get` and `unify` instructions of the WAM and these are carried over in spirit to our machine. However, some changes are necessitated by the need to deal with a richer class of terms and also a different representation for these. The instruction most affected by these factors is `get_structure F,Ai`. We use this instruction in compiling unification of an argument provided in `Ai` with a rigid term (or structure) of atomic type whose head is `F`. The first action to be carried out by this instruction in our context is to head-normalize the argument term. In the case that the result is a variable, a structure must be created on the heap and the variable bound to it[8]. The instruction anticipates the structure arguments and pushes a certain number of *capp* cells onto the heap before writing `F`. To facilitate this action, the instruction in our machine includes an extra argument. Thus, its actual form is `get_structure F,Ai,m`, where `m` indicates the arity of `F`[9]. In the case that the head-normalized version of `Ai` is a rigid structure, its head can be matched with `F`. Further, while the arguments of the structure may not appear in contiguous locations, our head-normalization procedure leaves references to these in such an arrangement in the SL stack. Thus by changing the `S` register into an index into the SL stack, the processing can be continued as in the WAM. Finally, there is one case for `Ai` that does not arise in the first-order case but may in the current context: `Ai` may be a pointer to a flexible non-variable term. In this case, a pair consisting of the contents of `Ai` and the address of the top of the heap is pushed onto the DS stack and added to the live list and execution continues in write mode.

The WAM instructions `get_constant C,Ai` and `unify_constant C` are affected in a manner similar to `get_structure` when carried over to our machine; in both cases a term needs to be head-normalized and a pair may have to be added to the active disagreement set. The required changes are entirely obvious at this point. Amongst the remaining instructions, `get_variable` and `unify_variable` are included unchanged and `simpl` is used in place of `get_value`. The two other *unify* instructions, `unify_value` and `unify_local_value` are also included and remain unchanged in write mode but behave like `simpl` in read mode.

The instructions described here, while catering largely to first-order unification, permit compilation of some aspects of higher-order unification as well. For example, the `get_variable` instruction can be used to bind a function or predicate variable. Notice that to permit such a use it is desirable to retain variables in clauses in an η-normal form, *i.e.*, with no surrounding η-redexes, but this is easily achieved by a preprocessing step prior to compilation. It is also possible to extend ideas of compilation to truly higher-order terms. For instance, the observation that unifying two rigid (de Bruijn) terms $\lambda \ldots \lambda\,(c\ s_1\ \ldots\ s_m)$ and $\lambda \ldots \lambda\,(c\ t_1\ \ldots\ t_m)$ can be reduced to unifying the set $\{\langle s_1, t_1\rangle, \ldots, \langle s_m, t_m\rangle\}$ can be used as the basis of an extension to `get_structure`. However, such an extension interferes with the annotations on terms and does not extend gracefully to a polymorphic context. Further, whether such extensions have a significant payoff remains unclear [22]. These ideas are therefore not currently incorporated into our implementation.

[8] In reality, a sound implementation must initiate an "occurs-check" here and our implementation does. However, we do not discuss this issue explicitly since our present purpose is largely to indicate the possibility of adapting the compilation scheme of the WAM.

[9] Note that this instruction is only used to compile unification with complex terms of *atomic* type. In a polymorphic situation, this implies that the target type of the structure should not be a variable.

The final aspect of compiling a clause involves generating code for the goal that constitutes its body. The compilation of complex goals and flexible goals is treated below. A rigid atomic goal is compiled into an invocation of the code for its name. This is as in the WAM and an identical set of instructions, with similar effects with regard to environment trimming, are used in our machine. Prior to this invocation, the argument registers have to be set up. This is achieved through the *put* instructions in the WAM and we import them to our context as well. Specifically, the `put_value` and `put_local_value` instructions are incorporated unchanged, and the `put_variable` instruction is changed to include as an additional argument the type of the variable being created. Notice that the instructions for creating lambda terms discussed earlier are also useful for this purpose. Given these instructions, the `put_structure` instruction of the WAM becomes redundant since `put_const`, `put_capp` and `put_fapp` suffice for the creation of structures.

The compilation scheme discussed here is based on a structure copying approach. In addition to the usual arguments for this approach, we have observed that it is required by our destructive implementation of reduction.

8.2 Compiling Complex and Flexible Goals

When the structure of a complex goal is known statically, the desired action can be compiled. Conjunctive and disjunctive goals are permitted even within the usual first-order language and an identical treatment is adopted for these in our implementation: conjunction is compiled into the sequential execution of a code for simpler goals (facilitated by the use of the continuation pointer, `CP`) and disjunction results in the creation of a choice point. Finally, an existential goal appearing in the body of a clause can be treated by the usual devices by noting the the classical equivalence of the formulas $\forall x\,(G \supset A)$ and $(\exists x\, G \supset A)$[10].

With regard to flexible goals, the main problem is that their top-level structure is determined dynamically, and so the specific action to be performed is not known at compilation time. Nevertheless, some part of the action can be compiled by using the knowledge of the possible cases that can arise. In particular, flexible goals are compiled into calls to a special procedure named *solve* to which the instantiated version of the goal is provided as an argument. In the case that (the normalized form of) the instantiated goal has a complex structure, the behavior of *solve* can be envisaged as if it were based on a compilation of the following clauses:

$(solve\ (G1 \wedge G2))$ `:-` $(solve\ G1)$ `,` $(solve\ G2)$.
$(solve\ (G1 \vee G2))$ `:-` $(solve\ G1)$ `;` $(solve\ G2)$.
$(solve\ (\exists\ G))$ `:-` $(solve\ (G\ X))$.

We use semicolon as the symbol for disjunction in the body of a clause here. In the remaining cases the action is as follows. If the argument is $\top$, `solve` succeeds immediately. If the argument is a flexible goal, then `solve` succeeds after instantiating the head of the goal to a term of the form $\lambda \ldots \lambda\, \top$, the binder being chosen based on type considerations. The last case is when the goal is of a rigid atomic kind. In this case, its arguments are loaded into appropriate argument registers and the head is used to determine the code to be invoked next.

[10]One case not mentioned explicitly is that of the atomic goal $\top$. This goal is really a "no-op" and is treated as such; its presence in the language is largely necessary for the proper treatment of uninstantiated flexible goals

8.3 Examples

We consider first the compilation of the *mapfun* predicate whose definition is reproduced below:

$(mapfun\ []\ F\ [])$.
$(mapfun\ [X|L1]\ F\ [(F\ X)|L2])$ `:-` $(mapfun\ L1\ F\ L2)$.

The code resulting from these clauses is shown below. The `switch_on_term` instruction used in this code is like the WAM instruction except that the first argument corresponds to the case when register `A1` points to a flexible term. Also, the `get_nil` and `get_list` instructions used here are like those in the WAM, *i.e.*, they are special cases of the `get_const` and `get_structure` instructions.

```
mapfun/3:  switch_on_term C1a, C1, C2, fail
C1a:       try_me_else C2a                  % mapfun
C1:        get_nil A1                       % []
           get_nil A3                       % F, []
           finish_unify                     %
           proceed                          %
C2a:       trust_me_else fail               % mapfun
C2:        get_list A1                      % [
           unify_variable X4                % X |
           unify_variable A1                % L1], F,
           get_list A3                      % [
           unify_variable X5                % S1 |
           unify_variable X3                % L2]
           put_unsafe_value A2,A2           % S1 = (F
           put_capp A6,A2,A4                % X
           simpl X5,X6                      % )
           finish_unify                     %
           execute mapfun/3                 % :- (mapfun L1 F L2)
```

The reader might use this code to solve the compiled version of the query

`?-` $(mapfun\ [a,b]\ G\ [(G\ b),(h\ a\ b)])$.

discussed in the previous section. At the end of the first phase of processing, much of the unification problem will be solved through the compiled code as anticipated, and the active disagreement set will contain the pair $\langle (G\ a),(G\ b)\rangle$. The goal $(mapfun\ [b]\ G\ [(h\ a\ b)])$ will then be invoked. At the time the penultimate instruction, *i.e.*, the `finish_unify` instruction, is invoked, the active disagreement set will contain the pairs $\langle (G\ a),(G\ b)\rangle$ and $\langle (G\ b),(h\ a\ b)\rangle$. Depending on the order in which imitation and projection substitutions are tried, up to 3 branch points might be created by the interpretive phase of higher-order unification, the binding $\lambda x\,(h\ a\ b)$ would be made to G and the live disagreement set will be reduced to the empty set. Finally, the goal $(mapfun\ []\ (\lambda x\,(h\ a\ b))\ [])$ will be invoked and will succeed. An attempt to find additional solutions will fail.

To illustrate the treatment of flexible goals, we consider the compilation of the clauses for *mappred* that are reproduced below:

$(mappred\ []\ P\ [])$.
$(mappred\ [X|L1]\ P\ [Y|L2])$ `:-` $(P\ X\ Y), (mappred\ L1\ P\ L2)$.

The code generated in this case is the following:

```
mappred/3:  switch_on_term C1a, C1, C2, fail
C1a:        try_me_else C2a                 % mappred
C1:         get_nil A1                      % []
            get_nil A3                      % P, []
            finish_unify                    %
            proceed                         %
C2a:        trust_me_else fail              % mappred
C2:         allocate                        %
            get_list A1                     % [
            unify_variable X4               % X |
            unify_variable Y1               % L1],
            get_variable Y2,A2              % P
            get_list A3                     % [
            unify_variable X5               % Y|
            unify_variable Y3               % L2]
            finish_unify                    %
            put_unsafe_value Y2,A2          % :- (P
            put_capp X1,A2,X4               % X
            put_capp A1,A1,X5               % Y
            call solve,3                    % )
            put_value Y1,A1                 % (mappred L1
            put_value Y2,A2                 % P
            put_value Y3,A3                 % L2
            execute mappred/3               % )
```

As an example of the use of this code, consider the query

`?-` $(mappred\ [bob, sue]\ (\lambda x\, \lambda y\, \exists z\, ((parent\ x\ z) \wedge (parent\ z\ y)))\ L)$.

from Section 2. We assume that our program contains the clauses that were presented for the *parent* predicate and also use the natural typing for the various constant and predicate symbols. The unification phase succeeds directly in the first call to *mappred*, binding P to $\lambda x\, \lambda y\, \exists z\, ((parent\ x\ z) \wedge (parent\ z\ y)$ and X to *bob*. The procedure `solve` is then invoked with the argument (in head-normal form) $\exists z\, ((parent\ bob\ z) \wedge (parent\ z\ Y))$. The code for `solve` simplifies this goal, eventually invoking the goals $(parent\ bob\ Z)$ and, after a binding is made to Z, $(parent\ john\ Y)$. A similar pattern results from the second call to *mappred* and original query eventually succeeds with the binding $[mary, kate]$ for L. Notice that all the unification problems that arise in this example are solved trivially without the use of imitations or projections. However, lambda conversion is used to produce goals of the appropriate shape here.

9 Conclusion

We have considered an enrichment to logic programming in this paper that is based on permitting a quantification over predicate and function symbols and on using lambda terms as data structures instead of first-order terms. We have motivated the extensions considered. Briefly, the new logical features support aspects of higher-order programming and considerably enhance the metalanguage abilities of logic programming. Our main concern here has been with the implementation problems that arise from the changes to the underlying logic. There are three main issues to be considered in this respect:

(i) A richer class of terms needs to be represented. Furthermore, the representation of these terms must support complex reduction operations on these terms.

(ii) A procedure for solving a more complicated kind of unification problem must be embedded in the implementation. The nature of the problem necessitates a treatment of branching within unification. Further, the attempt to solve a unification problem must at times be suspended and the problem must be carried along to some future point in the computation.

(iii) A method must be provided for handling goals whose structure might change in the course of execution.

We have presented solutions to these problems and have described an abstract machine that embeds these solutions. We have presented a notation for lambda terms that uses the de Bruijn scheme for eliminating the names of bound variables and also permits reduction substitutions to be performed lazily. This notation eliminates the need for α-conversions and also supports an efficient implementation of β-reduction. We have described further improvements to this notation and have also outlined a graph based reduction procedure based on it. With regard to the unification problem, we have described the notion of a branch point record that is analogous to the choice point record of the WAM but that supports branching in unification. We have also presented a method for representing (suspended) unification problems explicitly. One major concern is that of implementing first-order like unification efficiently, and this is achieved within our overall scheme. Specifically, we have adapted the compilation ideas of the WAM to deal with this kind of unification and have built devices into our machine that support this need within the interpretive phase of higher-order unification. Finally, we have indicated how flexible goals might be handled within our compilation scheme.

This paper represents, to our knowledge, the first detailed consideration of a machine for implementing a higher-order language such as ours. The work contained herein is important in that it highlights the issues involved and describes a basic structure for this machine that we believe will endure changes in detail. There is, nevertheless, considerable scope for improvement at the latter level. We have, for instance, not considered seriously the compilation of genuinely higher-order unification within our present model. Another issue that needs closer attention is the implementation of the occurs-check. There are several situations in which this can be safely omitted, and this aspect is built into the compiled code. However, we have not considered a scheme for recognizing these situations within the interpretive phase of unification. There is a mention of such a possibility in [4], and there appears to be benefit in looking into this issue more closely. A further issue is the interaction between β-reduction and backtracking. Within the current scheme, reductions are done only on demand. While this is clearly the preferred course, there are situations, as pointed out in [5], in which the need to undo reductions upon backtracking might result in extra work. It is of interest to see if these situations can be anticipated during compilation and the creation of a choice point delayed till after the reduction is performed. In a similar spirit, it appears worthwhile to examine the interaction between instructions in a segment of compiled code with regard to the creation of the matrices of terms in the SL stack. To provide a specific example, there are several instructions in the code presented for $mapfun$ in Section 8 that require the same structure to be created in the SL stack. If this can be anticipated during compilation, there is a possibility for sharing the work.

While there are improvements that can be made to our general scheme, we believe a better understanding of these possibilities must first be obtained from a working implementation. In this regard we note that while the focus in this paper has been on a simply typed, higher-order version of Horn clauses, our ultimate objective to provide an implementation of a language that includes universal quantification and implication in goals and also incorporates a form of polymorphic typing. We have considered the latter aspects in other work [21, 29] and have combined our various ideas into the design of a machine for the complete language [20]. The task of realizing this machine in code is currently underway.

Acknowledgements

Work on this paper has been supported by NSF Grant CCR-89-05825.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46. ACM Press, January 1990.

[2] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[3] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[4] Pascal Brisset and Olivier Ridoux. Naive reverse can be linear. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 857–870, Paris, France, June 1991. MIT Press.

[5] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Conference Record of the Workshop on the λProlog Programming Language*, Philadelphia, July-August 1992.

[6] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[7] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[8] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.

[9] G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

[10] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

[11] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[12] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, August 1989.

[13] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *Ninth International Conference on Automated Deduction*, pages 61–80, Argonne, IL, May 1988. Springer-Verlag.

[14] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–15. ACM Press, January 1990.

[15] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Logic Programming Conference*, pages 942–959, Seattle, Washington, August 1988. MIT Press.

[16] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, August 1990.

[17] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[18] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[19] Bharat Jayaraman and Gopalan Nadathur. Implementation techniques for scoping constructs in logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 871–886, Paris, France, June 1991. MIT Press.

[20] Keehang Kwon and Gopalan Nadathur. An instruction set for higher-order hereditary Harrop formulas. In *Proceedings of the Workshop on the λProlog Programming Language*, Philadelphia, 1992. (to appear).

[21] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. Submitted, August 1992.

[22] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Conference Record of the Workshop on the λProlog Programming Language*, Philadelphia, July-August 1992.

[23] Dale Miller. Hereditary Harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, pages 153–156, Moscow, August 1987.

[24] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[25] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 247–255. Association for Computational Linguistics, Morristown, New Jersey, 1986.

[26] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[27] Dale Miller and Gopalan Nadathur. λProlog version 2.7. Distributed in C-Prolog and Quintus Prolog source code, August 1988.

[28] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[29] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Technical Report CS-1993-17, Department of Computer Science, Duke University, July 1993.

[30] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[31] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[32] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[33] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, 1990.

[34] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms I: A generalization of environments. Manuscript in preparation, August 1993.

[35] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms II: Refinements and applications. Manuscript in preparation, August 1993.

[36] Lawrence R. Paulson. The representation of logics in higher-order logic. Technical Report Number 113, University of Cambridge, Computer Laboratory, August 1987.

[37] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 153–163, 1988.

[38] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[39] D.H.D. Warren. An abstract Prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.